Efficient Fault-Injection-based Assessment of Software-Implemented Hardware Fault Tolerance

Dissertation

zur Erlangung des Grades eines

D o k t o r s d e r I n g e n i e u r w i s s e n s c h a f t e n

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Horst Benjamin Schirmeier

Dortmund

2016

Tag der mündlichen Prüfung:	13. Juli 2016
Dekan:	Prof. Dr.-Ing. Gernot Fink
Gutachter:	Prof. Dr.-Ing. Olaf Spinczyk
	Prof. Dr. Andreas Polze

ABSTRACT

With continuously shrinking semiconductor structure sizes and lower supply voltages, the per-device susceptibility to transient and permanent hardware faults is on the rise. A class of countermeasures with growing popularity is *Software-Implemented Hardware Fault Tolerance* (SIHFT), which avoids expensive hardware mechanisms and can be applied application-specifically. However, SIHFT can, against intuition, cause more harm than good, because its overhead in execution time and memory space also increases the figurative "attack surface" of the system – it turns out that application-specific configuration of SIHFT is in fact a necessity rather than just an advantage.

Consequently, target programs need to be *analyzed* for particularly critical spots to harden. SIHFT-hardened programs need to be *measured and compared* throughout all development phases of the program to observe reliability improvements or deteriorations over time. Additionally, SIHFT implementations need to be *tested*.

The contributions of this dissertation focus on *Fault Injection* (FI) as an assessment technique satisfying all these requirements – *analysis*, *measurement and comparison*, and *test*. I describe the design and implementation of an FI tool, named Fail*, that overcomes several shortcomings in the state of the art, and enables research on the general drawbacks of simulation-based FI. As demonstrated in four case studies in the context of SIHFT research, Fail* provides novel fine-grained analysis techniques that exploit the newly gained possibility to analyze FI results from *complete* fault-space exploration. These analysis techniques aid SIHFT design decisions on the level of program modules, functions, variables, source-code lines, or single machine instructions.

Based on the experience from the case studies, I address the problem of large computation efforts that accompany exhaustive fault-space exploration from two different angles: Firstly, I develop a heuristical fault-space pruning technique that allows to freely trade the total FI-experiment count for result accuracy, while still providing information on *all* possible fault-space coordinates. Secondly, I speed up individual TAP-based FI experiments by improving the fast-forwarding operation by several orders of magnitude for most workloads. Finally, I dissect current practices in FI-based evaluation of SIHFT-hardened programs, identify three widespread pitfalls in the result interpretation, and advance the state of the art by defining a novel comparison metric.

PUBLICATIONS

Parts of this thesis have been published in the following peer-reviewed journal, conference and international workshop papers:

Horst Schirmeier, Rüdiger Kapitza, Daniel Lohmann, and Olaf Spinczyk. DanceOS: Towards dependability aspects in configurable embedded operating systems. In Alex Orailoglu, editor, *Proceedings of the 3rd HiPEAC Workshop on Design for Reliability (DFR '11)*, pages 21–26, Heraklion, Greece, January 2011. [SKLS11]

Horst Schirmeier, Jens Neuhalfen, Ingo Korb, Olaf Spinczyk, and Michael Engel. RAMpage: Graceful degradation management for memory errors in commodity Linux servers. In *Proceedings of the 17th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC '11)*, pages 89–98, Pasadena, CA, USA, December 2011. IEEE Computer Society Press. doi: 10.1109/PRDC.2011.20. [SNK+11]

Horst Schirmeier, Martin Hoffmann, Rüdiger Kapitza, Daniel Lohmann, and Olaf Spinczyk. Revisiting fault-injection experiment-platform architectures. In *Proceedings of the 17th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC '11)*, pages 284–285, Pasadena, CA, USA, December 2011. IEEE Computer Society Press. Fast abstract. doi: 10.1109/PRDC.2011.46. [SHK+11]

Horst Schirmeier, Martin Hoffmann, Rüdiger Kapitza, Daniel Lohmann, and Olaf Spinczyk. FAIL*: Towards a versatile fault-injection experiment framework. In Gero Mühl, Jan Richling, and Andreas Herkersdorf, editors, *25th International Conference on Architecture of Computing Systems (ARCS '12), Workshop Proceedings*, volume 200 of *Lecture Notes in Informatics*, pages 201–210. German Society of Informatics, March 2012. [SHK+12]

Christoph Borchert, **Horst Schirmeier**, and Olaf Spinczyk. Protecting the dynamic dispatch in C++ by dependability aspects. In *Proceedings of the 1st GI Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES '12)*, Lecture Notes in Informatics, pages 521–535. German Society of Informatics, September 2012. [BSS12]

Björn Döbel, **Horst Schirmeier**, and Michael Engel. Investigating the limitations of PVF for realistic program vulnerability assessment. In *Proceedings of the 5th HiPEAC Workshop on Design for Reliability (DFR '13)*, Berlin, Germany, January 2013. [DSE13]

Horst Schirmeier, Ingo Korb, Olaf Spinczyk, and Michael Engel. Efficient online memory error assessment and circumvention for Linux with RAMpage. *International Journal of Critical Computer-Based Systems*, 4(3):227–247, 2013. Special Issue on PRDC 2011 Dependable Architecture and Analysis. doi: 10.1504/IJCCBS.2013.058397. [SKSE13]

Christoph Borchert, **Horst Schirmeier**, and Olaf Spinczyk. Generative software-based memory error detection and correction for operating system data structures.

In *Proceedings of the 43rd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '13)*, Washington, DC, USA, June 2013. IEEE Computer Society Press. doi: 10.1109/DSN.2013.6575308. [BSS13a]

Christoph Borchert, **Horst Schirmeier**, and Olaf Spinczyk. Return-address protection in C/C++ code by dependability aspects. In *Proceedings of the 2nd GI Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES '13)*, Lecture Notes in Informatics. German Society of Informatics, September 2013. [BSS13b]

Martin Hoffmann, Peter Ulbrich, Christian Dietrich, **Horst Schirmeier**, Daniel Lohmann, and Wolfgang Schröder-Preikschat. A practitioner's guide to software-based soft-error mitigation using AN-codes. In *Proceedings of the 15th IEEE International Symposium on High Assurance Systems Engineering (HASE '14)*, pages 33–40, Miami, Florida, USA, January 2014. IEEE Computer Society Press. doi: 10.1109/HASE.2014.14. [HUD$^+$14b]

Horst Schirmeier, Lars Rademacher, and Olaf Spinczyk. Smart-hopping: Highly efficient ISA-level fault injection on real hardware. In *Proceedings of the 19th IEEE European Test Symposium (ETS '14)*, pages 69–74. IEEE Computer Society Press, May 2014. doi: 10.1109/ETS.2014.6847803. [SRS14]

Martin Hoffmann, Christoph Borchert, Christian Dietrich, **Horst Schirmeier**, Rüdiger Kapitza, Olaf Spinczyk, and Daniel Lohmann. Effectiveness of fault detection mechanisms in static and dynamic operating system designs. In *Proceedings of the 17th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '14)*, pages 230–237. IEEE Computer Society Press, June 2014. doi: 10.1109/ISORC.2014.26. [HBD$^+$14]

Horst Schirmeier, Christoph Borchert, and Olaf Spinczyk. Rapid fault-space exploration by evolutionary pruning. In *Proceedings of the 33rd International Conference on Computer Safety, Reliability and Security (SAFECOMP '14)*, Lecture Notes in Computer Science, pages 17–32. Springer-Verlag, September 2014. doi: 10.1007/978-3-319-10506-2_2. [SBS14]

Martin Hoffmann, Peter Ulbrich, Christian Dietrich, **Horst Schirmeier**, Daniel Lohmann, and Wolfgang Schröder-Preikschat. Experiences with software-based soft-error mitigation using AN codes. *Software Quality Journal*, pages 1–27, November 2014. doi: 10.1007/s11219-014-9260-4. [HUD$^+$14a]

Horst Schirmeier, Christoph Borchert, and Olaf Spinczyk. Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors. In *Proceedings of the 45th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '15)*, pages 319–330, Washington, DC, USA, June 2015. IEEE Computer Society Press. doi: 10.1109/DSN.2015.44. [SBS15]

Christoph Borchert, **Horst Schirmeier**, and Olaf Spinczyk. Generic soft-error detection and correction for concurrent data structures. *IEEE Transactions on Dependable and Secure Computing*, PP(99), 2015. Pre-print. doi: 10.1109/TDSC.2015.2427832. [BSS15]

Horst Schirmeier, Martin Hoffmann, Christian Dietrich, Michael Lenz, Daniel Lohmann, and Olaf Spinczyk. FAIL*: An open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance. In *Proceedings of the 11th European Dependable Computing Conference (EDCC '15)*, pages 245–255. IEEE Computer Society Press, September 2015. doi: 10.1109/EDCC.2015.28. [SHD$^+$15]

Thiago Santini, Christoph Borchert, Christian Dietrich, **Horst Schirmeier**, Martin Hoffmann, Olaf Spinczyk, Daniel Lohmann, Flávio Rech Wagner, and Paolo Rech. Evaluating the radiation reliability of dependability-oriented real-time operating systems. In *Proceedings of the 12th Workshop on Silicon Errors in Logic – System Effects (SELSE '16)*, March 2016. [SBD$^+$16]

ACKNOWLEDGMENTS

First and foremost I want to thank my advisor Prof. Dr.-Ing. Olaf Spinczyk, who has repeatedly encouraged me during our time in Erlangen to follow him to Dortmund, and to start working in research. Without his continued support, advice, and funding, this thesis would not exist. I would also like to thank Prof. Dr. Andreas Polze for his time and commitment to review this thesis.

Over the years of researching, teaching, publishing, and networking, I met and collaborated with many excellent people. Jochen Streicher has been sharing my office since the first day, and I guess every single part of my work has been influenced by some piece of advice or a result of a discussion with him. Michael Engel always provided advice, and words of encouragement when I needed them.

In the context of the SPP-1500 *DanceOS* project,[1] I had the pleasure to work with Daniel Lohmann and Rüdiger Kapitza, who helped focusing my research on what finally led to this dissertation.

Then there are, of course, all the FAIL* guys. Martin Hoffmann launched the first FAIL* prototype together with me, and he has since continuously contributed additional components, plugins, the T32 simulator back-end, or manpower in person of his student assistants. Among many other details, Christian Dietrich helped turning my assessment-cycle layer tool prototypes into the form they are in today. Christoph Borchert was among the first FAIL* users, and our profound discussions on fault tolerance and fault injection have been essential for many of the findings in this dissertation. Björn Döbel provided several really challenging use cases, resulting in several years of enjoyable collaboration and occasional late-night discussions at varying SPP meetings. My student assistants Adrian Böckenkamp, Richard Hellwig, and Michael Lenz also put lots of work into FAIL*, just as my students Tobias Friemel and Lars Rademacher. Without these contributors, FAIL* would not be where it is today – thank you, guys!

Additionally, I would like to thank several people for helpful – professional or personal – discussions and the occasional coffee-machine chatter, without which the LS12 respectively CS-faculty working environment would have been a different place: Alexander, Andreas, Björn, Boguslaw, Florian, Georg, Helena, Hendrik, Ingo (in particular also for his work on RAMpage), Jan, Kuan, Matthias, Markus, Olaf, Patrick, and Thomas.

Last, but certainly not least, I would like to thank my family – especially my parents for having supported me since day one, and my wife Steffi for coping with the by-products of my thesis writing, for hugs when needed the most, and for everything else.

1 This work was partly supported by the German Research Foundation (DFG) priority program SPP 1500 under grants no. SP 968/5-1, SP 968/5-2, and SP 968/5-3.

CONTENTS

ACRONYMS

ABFT	Algorithm-Based Fault Tolerance
ABS	Anti-lock Braking System
ACE	Architecturally Correct Execution
ALU	Arithmetic Logic Unit
ANN	Artificial Neural Network
AOP	Aspect-Oriented Programming
API	Application Programming Interface
AVF	Architectural Vulnerability Factor
AVI	Application Vulnerability Index
BB	Basic Block
BEEC	Block Entry Exit Checking
BER	Backward Error Recovery
BSSC	Block Signature Self Checking
CCA	Control-flow Checking using Assertions
CFE	Control-Flow Error
CFG	Control-Flow Graph
CFM	Control-Flow Monitoring
CMOS	Complementary Metal-Oxide-Semiconductor
CMP	Chip-level Multiprocessing
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DFT	Dilution Fault Tolerance
DMR	Dual-Modular Redundancy
DRAM	Dynamic Random-Access Memory
DUE	Detected Unrecoverable Error
DUEC	Def/Use Equivalence Class
ECC	Error-Correcting Code
ECCA	Enhanced CCA
EDC	Egregious Data Corruption
EDDI	Error Detection by Duplicated Instructions
EDM	Error-Detection Mechanism
EEA	Execution-Environment Abstraction
EF	Effective Number of Failures
EMI	Electromagnetic Interference
ERM	Error-Recovery Mechanism
FEC	Forward Error Correction
FER	Forward Error Recovery
FI	Fault Injection
FIT	Failure In Time
FPGA	Field-Programmable Gate Array
FSC	Fault-Similarity Class
FSP	Fault-Similarity Pruning
FVI	Function Vulnerability Index
GOP	Generic Object Protection
HCI	Hot Carrier Injection
IGFET	Insulated-Gate Field-Effect Transistor
IR	Intermediate Representation

ISA	Instruction-Set Architecture
IVI	Instruction Vulnerability Index
JIT	Just-In-Time
LANSCE	Los Alamos Neutron Science CEnter
MBU	Multiple-Bit Upset
MCU	Multiple-Cell Upset
MISFET	Metal-Insulator-Semiconductor Field-Effect Transistor
MITF	Mean Instructions To Failure
MMU	Memory-Management Unit
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor
MPU	Memory-Protection Unit
MTBF	Mean Time Between Failures
MTTF	Mean Time To Failure
MTTR	Mean Time To Repair
MWBF	Mean Workload Between Failures
MWTF	Mean Work To Failure
NBTI	Negative Bias Temperature Instability
NMR	N-Modular Redundancy
OCD	On-Chip Debugger
ODD	Optimum Data Duplication
PVF	Program Vulnerability Factor
RAP	Return-Address Protection
RECCO	REliable Code COmpiler
RMSE	Root Mean Squared Error
RMT	Redundant MultiThreading
RTL	Register-Transfer Level
SBU	Single-Bit Upset
SDC	Silent Data Corruption
SECDED	Single-Error Correct Double-Error Detect
SEE	Single-Event Effect
SER	Soft-Error Rate
SEU	Single Event Upset
SIHFT	Software-Implemented Hardware Fault Tolerance
SMP	Symmetric Multiprocessing
SMT	Simultaneous Multithreading
SOI	Silicon-On-Insulator
SoR	Sphere of Replication
SQL	Structured Query Language
SRAM	Static Random-Access Memory
SRT	Simultaneous and Redundantly Threaded
SWIFI	Software-Implemented Fault Injection
TAP	Test-Access Port
TMR	Triple-Modular Redundancy
TOCTTOU	Time-of-Check-to-Time-of-Use
TTF	Time To Failure
VM	Virtual Machine
WSOS	Winter School on Operating Systems

1 INTRODUCTION

Contents

In the decades since the invention of semiconductor-based integrated circuits [Kil76], computer systems have pervaded all areas of technology. Fueled by constantly progressing miniaturization [Int13] accompanied by plummeting per-transistor prizes [Sch97], and – compared to earlier tube-based solutions – massively lower weight and energy consumption, computers have become omnipresent even in safety domains such as automotive, avionics, or space flight. In these domains, computing hardware and software not only must satisfy common nonfunctional requirements on properties such as *cost* and *energy efficiency*, but also *timeliness* and especially *dependability*.

As chip technology is continuously moving towards higher densities and lower operating voltages [Int13], its sensitivity to hardware faults caused by electromagnetic radiation, interfering impulses, aging and thermal effects, and process variations also increases dramatically [Con02, SKK+02, Con03, Con05, Bor05, Bau05b, NX06, DYdS+10, DW11]. Consequently, every new generation of hardware designs for embedded systems exhibits an increasing rate of permanent (remaining for indefinite periods), intermittent (reappearing after their first occurrence), and transient (appearing, and disappearing) hardware faults [Muk08].

Chapter 2 sheds more light on causes and effects of hardware faults.

Today, the magnitude of these effects are far from purely academic but can be observed in all areas of computing. For example, Santini et al. [SRCRW15] estimate 175,000 user-observable errors per year on iPhone 3 devices used aboard airplanes, caused by cosmic radiation. With much more severe outcomes – including at least 89 casualties [CBS10], and monetary losses beyond one billion U.S. dollars for the car manufacturer, – several Toyota car models were reported to rarely accelerate unintendedly in the years between 2000 to 2010. Although the exact cause could not be proven, radiation-induced bit flips – and a lack of protection against them – were assumed to be probable culprits [Yos13, Wik13]. But even outside the world of embedded systems, for example in data centers [HSS12] or large-scale scientific computing [SG10], reports of hardware faults have accumulated over the years.

At the hardware-software boundary, the previously maintained illusion of always correctly functioning hardware vanishes step by step [HBB⁺11]. This trend towards what has been termed the "soft-error wall" [Muk08] or "reliability wall" [BJS07] creates new challenges for the development of reliable embedded systems, such as automotive control units. As a reaction to these new threats, certification authorities demand explicit measures to cope with transient faults in their functional safety standards, such as IEC 61508 [IEC98] or ISO 26262 [ISO11]. However, for cost-sensitive mass products – such as cars – manufacturers cannot deploy full-fledged hardware redundancy mechanisms. Examples for such hardware mechanisms are triple-modular redundant components with an additional voter, as common in avionic systems, or memory protected with an *Error-Correcting Code* (ECC) [Muk08, RTSM11].

Before dissecting the state of the art in fault injection, and identifying gaps therein that lead to the contributions of this dissertation in Section 1.2, the following Section 1.1 provides the motivating context for this dissertation – *software-implemented* hardware fault tolerance. Section 1.3 provides an outline for this dissertation, and Section 1.4 gives an overview of my own contributions to research results obtained in cooperation with other researchers.

1.1 SOFTWARE-IMPLEMENTED HARDWARE FAULT TOLERANCE

A primary reason for the high cost of hardware-implemented fault tolerance is its generality, or, in other words, its obliviousness to application demands. For example, the redundancy necessary for the ECC protects memory cells regardless of the criticality of their contents for the application to function within parameters. This waste of resources cannot be avoided in *Commercial Off-The-Shelf* (COTS) systems if fault tolerance is implemented on the hardware level. Another argument against hardware-implemented fault tolerance is its severe impact on performance and energy consumption: For example, the developers of the ARGOS satellite concluded that their radiation-hardened hardware was simply too slow for the *"data processing job intended for it"* [LWW⁺02].

Instead, the problem has to be dealt with – at least partially – in the software [HBB⁺11]. To not diminish all gains from the aforementioned higher chip densities and lower operating voltages, embedded-software developers have to *selectively* place application-specific error detection [Hil00, HJS02a, OSM02, NV03, RCV$^+$05b, SHLR$^+$09, LCP$^+$09, HAN12] and recovery mechanisms [CCK$^+$06, PGZ08, BSS12, BSS13a, BSS13b, RMCJ13, HBD$^+$14] (EDMs/ ERMs) in their mixed-criticality systems. Critical tasks and sensitive spots in the software stack must be hardened against hardware faults, while the remaining – less critical – components economize resource consumption by occasionally tolerating incorrect results.

Section 2.4 explains more in detail how countermeasures against hardware faults work.

The necessity to detect and recover from faults on the software level is not a vague possibility for the future anymore. For example, the German semiconductor manufacturer Infineon Technologies AG has been advocating PRO-SIL™ [Kon08] – a library of SIHFT mechanisms – as a means complementary to the hardware mechanisms in their TriCore® embedded microcontroller architecture.

However, experience shows that the development and placement of EDMs/ ERMs is a difficult task in practice. Against intuition, software-based fault-tolerance measures often cause more harm than good [GSR05, MCR$^+$12, BSS13a, SRJW14, SBS15], as their overhead in time and space also increases the figurative "attack surface" of the system for adverse environmental effects. Besides the measures' effectiveness and efficiency, especially their placement in the target system – the choice of which data structures or program modules to protect, and which to leave unprotected – is essential for the overall system's resiliency. Furthermore, even small changes to the functional part of the software, such as refactoring a pointer-based linked list into an array-based algorithm, can have dramatic impact on the robustness.

Hence, analogous to common practices in iterative software development, a developer and deployer of fault-tolerance measures must be equipped with techniques and tools to analyze, test, measure, and compare EDM/ERM implementations in their target environment. *Analysis* yields detailed information on the fault sensitivity of specific parts of the program – for example the most critical variables, or program sections, – a basis for the informed placement of fault-tolerance measures. *Testing* allows to find implementation flaws once the measures are placed. *Measurement* in turn allows to quantitatively observe the improvement – or deterioration – of both effectiveness and efficiency of fault-tolerance measures, and is a direct prerequisite for *comparison* of different implementations.

1.2 FAULT INJECTION: TOOLS, TECHNIQUES, AND CONTRIBUTIONS OF THIS DISSERTATION

This dissertation focuses on analysis, testing, measurement, and comparison of hardware-fault resilience of software systems, in particular embedded system software. In this context, I mainly investigate the effect of *transient* hardware faults, also known as *Single Event Upsets* (SEUs) or *soft errors*, that manifest on the software level as single- or multi-bit faults – also known as *bit flips* – in main memory or in CPU registers. This section describes the problem statement and the chosen approach involving *Fault Injection* (FI) in detail, and explicitly lists the contributions of this dissertation, including

- the design and implementation of an FI tool that eliminates several shortcomings widespread in existing tools and the literature,
- two techniques to significantly speed up FI campaigns and experiments,
- a metric properly interpreting FI results for measurement and comparison, and
- new fine-grained analysis methods supporting the placement of EDMs/ ERMs.

1.2.1 *Fault Injection*

For at least two decades, the primary testing and measurement technique in the field of SIHFT, which has to a certain degree also been used for analysis and comparison, has been Fault Injection (FI) [AAA$^+$90, CP95, HTI97, ES98, BP03, ZAV04, Che10, KDN14]. Since the identification of primary and secondary physical causes for soft errors in the 1970s [BSH75, MW79], several techniques for the artificial injection of hardware faults have been devised. These techniques are used to mimic the effects of the original causes for hardware faults, but with extremely increased occurrence probability to trigger the fault-tolerance mechanisms often enough for sufficient evidence of their effectiveness.[1]

Chapter 3 discusses different FI techniques in detail, and describes existing tools implementing them.

Nowadays, so-called *simulation-based* FI [BP03, KDN14] is one of the most favored FI techniques. The approach to inject faults into purely simulated hardware offers many advantages:

- The target system software does not need any modifications, which induce a "probe effect" in other FI approaches.
- Faults can be injected with high precision regarding space – *where* to inject – and time – *when* to inject, – and can reach any part of the hardware that is actually simulated.
- Experiments can be repeated deterministically, which is important for many optimization techniques.
- An FI *campaign*, consisting of a large number of individual FI experiments, can be sped up by running many simulator instances in parallel, for example making use of a computing cluster.

Despite these advantages, simulation-based FI also comprises a distinct set of drawbacks:

1 Note that, besides hardware faults, FI is also used to inject other types of faults, such as software bugs – or, "software faults" [Voa97, VG00, MCV00, DM06, NCDM13], – which is outside the scope of this dissertation. Throughout this work, the term FI means the injection of *hardware* faults unless explicitly stated otherwise.

- Simulators are known to be very slow compared to the real hardware they simulate, especially in the case of detailed low-level simulations, resulting in long FI campaign runtimes.
- A simulator is only a – sometimes very abstract – approximation of the real hardware. Results obtained from simulations must be judged with the simulation accuracy in mind.

Before approaching the general problem of long campaign runtimes, I will first describe some basics in the context of simulation-based FI, and analyze shortcomings in the state of the art of simulation-based FI tools.

1.2.2 *Fault-Space Exploration and the Fault-Injection Tool Landscape*

As mentioned above, an FI *campaign* consists of many individual FI experiments, each running the analyzed target software – often called the *workload* – in a similar way. In each experiment run, usually *one* fault is injected at a specific point in time after the workload start – measured, for example, in CPU cycles, – and at a specific location – for example, flipping one specific bit in main memory. After injecting the fault, the workload continues running, and the FI tool observes its behavior and its reaction to the injected fault. Depending on this observation, the FI tool categorizes the experiment result – or *outcome* – according to a predefined *failure model*, for example:

NO EFFECT: The workload behaved the same way as in a run without FI, for example because the fault was masked by a subsequent operation.

DETECTED: An EDM detected the fault, and initiated a recovery procedure, for example a reboot.

SILENT DATA CORRUPTION (SDC): The workload's externalized state differed from the correct output, for example because the fault affected a calculation, and was not detected by an EDM.

TIMEOUT: After injecting the fault, the workload did not finish within a predefined time limit, for example because it faultily entered an infinite loop.

The difference between each experiment in an FI campaign is the point in time *when* a fault is injected, and the location *where* the fault is injected. These time and space coordinates span a so-called *fault space*, as depicted in Figure 1.1: Each coordinate, shown as a black dot, represents a possible individual FI experiment.

For detailed analyses, a *complete* fault-space exploration, yielding information on each and every possible fault-space coordinate, could be promising. Such detailed result data could be broken down to the level of single bits or individual program instructions to inform the developer about the criticality of single data objects, variables, modules, functions, source-code lines, or any other level of granularity deemed necessary for fault-tolerance related design decisions.

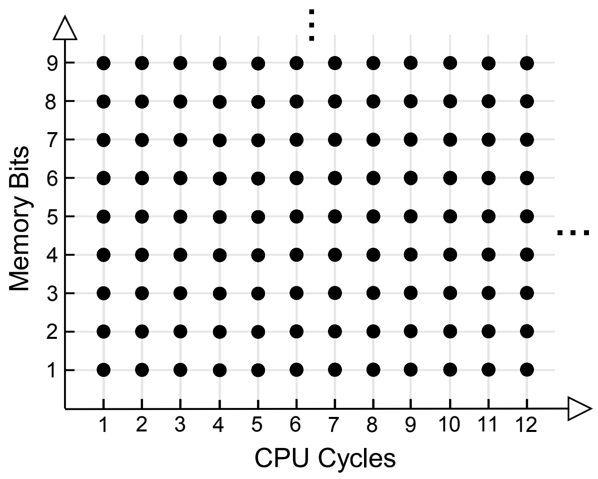


Figure 1.1: Single-bit flip fault space for a run-to-completion workload. Each dot represents a possible FI space/time coordinate. The workload proceeds until and stops at a specific point in time, the corresponding memory bit is flipped, and the workload resumes.

In spite of the potential, the state of the art in simulator-based FI tools only very recently started to provide complete fault-space exploration support in Relyzer [HANR12], or SmartInjector [LT13]. Nevertheless, these tools *do not introduce detailed analysis methods* that would make use of the detailed result data, but focus on methods how to obtain them. To my knowledge, all other FI tools in existence are only capable of randomly sampling the fault space, lacking the result detail for advanced analysis methods.

Separate from the drawbacks mentioned in the previous section, which are inherent to simulation-based FI in general, the existing FI tool landscape has several specific problems of its own:

- As explained above, all but the most recent [DBG+09, GDJ+11, HANR12, LT13] FI tools are incapable of complete fault-space exploration. There exists no FI tool that provides advanced analysis capabilities based on complete fault-space exploration down to the level of single data objects, variables, modules, functions, source-code lines, or any other level of granularity deemed necessary for fault-tolerance related design decisions.

- The vast majority of existing FI tools aims at *testing*, but lacks proper metrics for *measurement* and *comparison*. The latter is essential for driving design decisions in SIHFT.

- Most existing *simulators* do not have FI capabilities of their own. Researchers usually time-consumingly patch FI functionality into the code of an existing simulator [PTAB14], resulting in low code quality and maintainability issues. In some cases, they even write a completely new simulator with FI in mind [PSDC07, SPS09], spending resources on an engineering task that has already been solved by others, and also resulting in scantly maintainable simulation code that is heavily intertwined with FI code [SHK+11].

- A side effect of this problem is that most simulation-based FI tools are limited to one specific simulator. This inflexibility becomes relevant when the requirements, for example regarding the CPU architecture, change during development.
- Most simulation-based tools cannot be used for FI techniques other than simulation-based FI. This forces a costly switch to a separate tool when in later development stages, FI experiments are supposed to be repeated on real prototype hardware, using, for example, *test-port based* FI[2].
- Additionally, a practical observation is that most FI tools are not publicly available to other researchers for conducting own FI campaigns, or doing research on FI techniques. While Chapter 3 discusses possible reasons for this phenomenon, nevertheless its existence hinders own studies with and on FI.

The observation of these disadvantages in the tool landscape motivate the first contribution of this dissertation:

CONTRIBUTION 1: DESIGN AND IMPLEMENTATION OF FAIL*

The first contribution of this dissertation is the *design and implementation of an FI tool*, named FAIL*, that overcomes the aforementioned problems of existing tools in the field, and enables research on the general drawbacks of simulation-based FI.

- First and foremost, FAIL* supports complete fault-space exploration, and analysis methods utilizing the detail level of the resulting data. These features will be covered by the following, separately described contributions 2, 3, and 5.
- FAIL* explicitly supports measurement and comparison, and provides an appropriate metric for this purpose. This metric is separately described in contribution 4.
- By using minimally invasive software coupling techniques, FAIL* attaches to existing simulator software without provoking serious maintainability issues.
- With the help of a carefully designed abstraction layer, FAIL* can be – transparently to the experiment designer – used to inject faults in several target back ends (including both simulators and real prototype hardware) using two different FI techniques.

2 In test-port based FI, faults are injected into prototype hardware via a test access port that is commonly available in embedded microcontrollers, such as JTAG [MT90, SZR03], BDM [How96, RSR99], or Nexus [II99, YdAL+03, FGAF06]. See Chapter 3 for more details on this FI technique.

- Additionally, FAIL* is the basis for the other contributions of this dissertation, which were achieved by using and enhancing this tool.

Chapter 4 describes FAIL*'s design and implementation in detail.

1.2.3 *Fault-Injection Campaign Runtimes*

Another central problem of simulation-based FI is the amount of time and computing power that needs to be spent for FI campaigns, especially when aiming for complete fault-space exploration.

As explained in the previous section, for detailed analyses, information on each and every possible fault-space coordinate can be essential. In this case, the FI campaign exhaustively explores the fault space, running experiments for *every* coordinate. From Figure 1.1 it becomes clear that the fault-space size w – equaling the number of necessary experiments – grows quadratically with the amount of memory bits Δm it uses, and the workload's total runtime Δt: The size calculates as $w = \Delta t \cdot \Delta m$.

For a relatively small workload that runs for $1s$ or $\Delta t = 10^9$ cycles on a 1 GHz CPU, and uses 1 MiB or $\Delta m = 2^{23}$ bits of RAM, the fault-space has a size of $w = 10^9 \cdot 2^{23} \approx 8.4 \cdot 10^{15}$. If the experiment runtime $T_{\text{experiment}}$ for one simulation run is known, we can calculate the campaign's total runtime $T_{\text{campaign}} = w \cdot T_{\text{experiment}}$. Even if a single FI experiment – involving the simulation of 10^9 CPU cycles – only takes $T_{\text{experiment}} = 1$ s on the host CPU, the complete campaign would take $T_{\text{campaign}} = 8.4 \cdot 10^{15} \cdot 1$ s, or about 266 million CPU years. Actually running this campaign is clearly infeasible even with large amounts of computing power at hand – a problem we call the *fault-space explosion* [HUD+14b, HUD+14a, SBS15].

Clearly, T_{campaign} must be reduced by many orders of magnitude to reach feasible CPU runtimes. As the FI experiments within a campaign can be conducted independently of each other, an obvious approach to cut down the *wall-clock time* from start to end of the campaign is to run multiple experiments in parallel on available computing hardware. Nevertheless, this solution is strongly limited by the number of available host CPUs, and therefore cannot reduce the runtime by more than a few orders of magnitude.

The remaining options of reducing $T_{\text{campaign}} = w \cdot T_{\text{experiment}}$ are to either reduce the number of FI experiments N to an $N \ll w$, or to reduce the (average) runtime $T_{\text{experiment}}$ for an individual experiment [GRSRV06].

1.2.4 *Fault-Space Pruning*

Section 3.3.1 explains the def/use-pruning technique – exploiting the equivalence of different fault-space coordinates – in detail.

The first option, reducing the number of FI experiments, is a process called fault-list collapsing [BRIM98, BGC+02], fault pruning [HANR12], or *fault-space pruning* [SBS14]. A very effective and widespread fault-space pruning technique based on additional information on memory accesses of the workload is called *def/use pruning* [SJPB95, GS95]. It reduces the number of FI experiments by several orders of magnitude even without reducing precision.

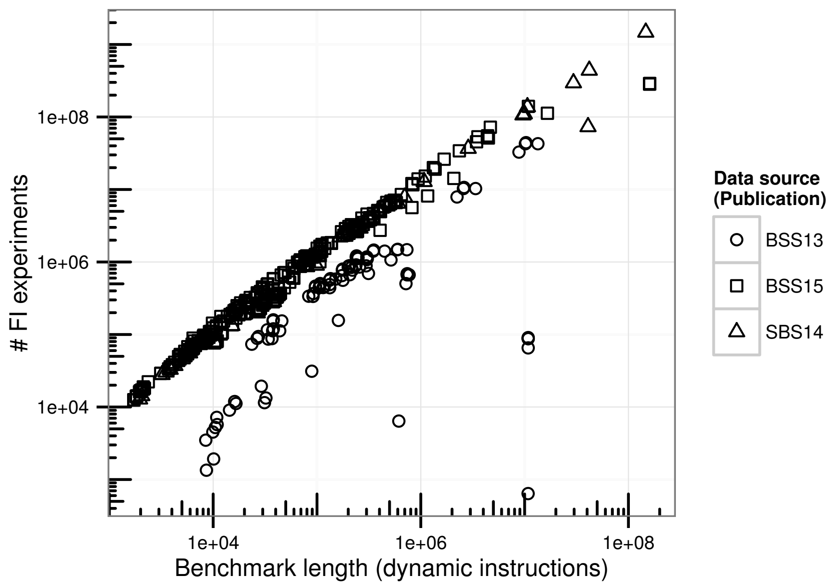


Figure 1.2: Workload lengths in dynamic instructions of benchmarks used in three different publications, and number of FI experiments after def/use fault-space pruning (log-log plot): The number of FI experiments necessary for an exhaustive fault-space exploration grows roughly linearly with the workload length – but is also influenced by the nature of the workload, explaining the outliers.

Using data from FI campaigns conducted for three of my publications [BSS13a, SBS14, BSS15], Figure 1.2 shows for each workload the length – in dynamic CPU instructions – and the number of FI experiments necessary for an exhaustive fault-space exploration *after def/use pruning*. The data shows that, for non-trivial workloads, the number of FI experiments is still large – with a maximum of about 1.5 billion experiments for one specific benchmark from [SBS14], – and grows about linearly with the number of dynamic instructions in the workload.

CONTRIBUTION 2: A HEURISTICAL FAULT-SPACE PRUNING TECHNIQUE

The second contribution of this dissertation directly addresses the problem of exhaustive fault-space exploration and the accompanying large number of necessary FI experiments:

- I contribute *a novel, heuristical fault-space pruning technique* called Fault-Similarity Pruning.
- The heuristic allows to freely trade the FI-experiment count for result accuracy, and – unlike prevalent sampling techniques – provides information on *all* possible fault-space coordinates.

Chapter 5 describes the *Fault-Similarity Pruning* (FSP) heuristic and its implementation in FAIL*, and evaluates it in detail.

1.2.5 *Reducing Experiment Runtime on Real Hardware*

As explained in Section 1.2.3, the second option of reducing $T_{\text{campaign}} = w \cdot T_{\text{experiment}}$ is to reduce the second multiplication factor, the average runtime $T_{\text{experiment}}$ for an individual experiment. This can be achieved by either reducing the number of workload instructions *actually executed* on the target – be it a simulator, or another execution environment, – or by increasing the target's execution speed.

Section 3.3.2 discusses related work on FI-experiment speedup.

The literature describes solutions to the *first* problem, reducing the number of executed instructions in the experiment phase before [PRSRV00b, PRSRV00a, BGC+02, HVAN14, PTAB14] and after [BGC+02, LT13, HVAN14] the fault is injected. Approaching the *second* problem by increasing the execution speed *for hardware simulators* is *not* the focus of this dissertation: FAIL* has been developed with loose coupling of exchangeable target back ends in order not to be distracted by reinventing or improving existing simulator technology.

> CONTRIBUTION 3: SMART-HOPPING – A FAST-FORWARDING TECHNIQUE FOR FAULT-INJECTION EXPERIMENTS ON REAL HARDWARE
>
> The third contribution of this dissertation addresses the runtime of individual FI experiments on real prototype hardware, controlled via a *Test-Access Port* (TAP):
>
> - The *fast-forwarding operation* – advancing a workload to the specific CPU cycle in its instruction stream when the fault is injected – is a major bottleneck in the runtime of an FI experiment running on TAP-controlled COTS prototype hardware.
> - Consequently, I contribute a novel trace-based *fast-forwarding technique for TAP-based FI* termed Smart-Hopping. The technique increases the speed of the fast-forwarding operation *by several orders of magnitude* for most workloads.

Chapter 6 describes Smart-Hopping in detail, explains how it blends into the FAIL* infrastructure, and evaluates it with a widely used benchmark suite.

1.2.6 *Measuring and Comparing Program Susceptibility to Soft Errors*

The remaining two contributions of this dissertation address the state of the art on interpreting, analyzing, and presenting the *results* obtained from FI campaigns.

Section 3.1.2.5 discusses the fault-coverage factor metric more in detail.

In the overwhelming majority of existing publications describing SIHFT solutions (for example, [Fuc96, RSRVT01, OSM02, NSV04, RCV+05b, CRA06, CCK+06, BSS12, HAN12]), program susceptibility to soft errors is measured and compared using the *fault-coverage factor* – or short: fault coverage – metric [BCS69]. This metric, *"defined as the probability of system recovery given that a fault exists"* [PMAC95], can be calculated with only a sample of the fault space,[3] and reduces the FI results to percentages of each outcome defined in the failure model (see Section 1.2.2). For example, if $F = 80$ out of $N = 1000$ sampled FI experiments yield a result where the fault was neither detected nor masked – in the example of Section 1.2.2, either Silent Data Corruption (SDC) or timeout, – the fault coverage c can be calculated as

$$c = 1 - \frac{F}{N} = 1 - \frac{80}{1000} = 92\,\%.$$

CONTRIBUTION 4: A COMPARISON METRIC FOR SOFT-ERROR SUSCEPTIBILITY OF PROGRAMS

The fourth contribution of this dissertation addresses this and several other problematic measurement practices in the field of SIHFT:

- After uncovering another two ill-advised practices in the interpretation of FI results, I demonstrate that *the fault-coverage factor metric is unsound* for the comparison of soft-error susceptibility of programs, and can lead to wrong design decisions.
- These findings are substantiated by an illustrative thought experiment and results from an FI campaign conducted with FAIL*.
- Based on these insights, *I construct an objective metric usable for comparison* using extrapolated absolute failure counts, and introduce the mathematical foundation supporting this proposition.

Chapter 7 analyzes common measurement and comparison practices in the field in detail, reveals the unsoundness of the fault-coverage metric for a widely-used purpose, and provides the mathematical foundation for the new comparison metric.

1.2.7 *Fine-Grained Post-Injection Analysis and Result Visualization*

With a metric at hand to measure and compare the *global* susceptibility of programs to soft errors, the fifth and final contribution of this dissertation addresses fine-grained analysis and result visualization on the *local* level. As explained in Section 1.2.2, the state of the art in FI tools presents a prominent gap by lacking advanced analysis capabilities based on complete fault-space exploration. This dissertation takes first steps to gauge this dormant potential.

Several researchers already addressed the problem of locating particularly vulnerable – or *critical* – spots of the program [HJS01, HJS02a, BDCDN+03, HJS04, PKI05, PKI07, PNKI08, SK08b, SK09b, SSSF10b, PSC+11, PKI11, LJ11, HAN12, PNKI13]. These critical spots are either logical candidates for the placement of EDMs/ERMs, or drive other dependability-related design decisions – like choosing a different algorithm for a particular functional requirement. Most of these existing works are specific to certain classes of EDMs/

Section 3.4 discusses fine-grained criticality-analysis methods in detail.

3 Of course, results from exploring the complete fault space can also be used, but are unnecessarily costly if only the fault coverage is calculated.

ERMs, or limited to one specific, sometimes very coarse level of granularity – which already strongly predetermines and limits the analysis results, and, consequently, possible placements. Almost all use the aforementioned, problematic fault-coverage factor metric for either measuring criticality, or for the evaluation of their approach after the placement of EDMs/ERMs in the spots deemed most critical.

> CONTRIBUTION 5: FAULT-INJECTION RESULT ANALYSIS
>
> Hence, the fifth and last contribution of this dissertation introduces *analysis methods at different levels of granularity, operating on the results from complete fault-space exploration.*
>
> - By collecting data on the whole fault space, the decision on the analysis granularity in space and time can be postponed until after the FI campaign.
> - The detail level of the result data allows to run analyses aiding design decisions on the level of variables, program phases, modules, functions, source-code lines, or single instructions, and opens up potential for more analysis types in the future.

The FAIL* case studies in Chapter 4 take first steps at gauging the dormant potential of analyses of FI results from complete fault-space exploration, and present different analysis methods.

1.3 OUTLINE

This section gives an outline of this dissertation, and assigns the five contributions from Section 1.2 to chapters.

- Chapter 2 provides more background by shedding light on the causes and effects of hardware faults, and defines basic terms and metrics. The chapter also describes possible countermeasures on both the hardware and software levels, with a focus on the basic principles of *software*-implemented hardware fault tolerance (SIHFT) and *soft errors.*
- Chapter 3 surveys the state of the art in hardware FI: First, the chapter discusses the possible goals of injecting hardware faults – and which of these this dissertation focuses on, – and how faults from the real world are mapped to artificial injection using *fault models.* Subsequently, further basic terms and definitions in the context of FI are introduced, and an overview on the FI tool landscape is provided. After drawing conclusions on the capabilities and properties of existing tools, I describe optimization techniques from the literature that help reducing the efforts spent in FI campaigns. The chapter ends with a discussion of metrics used in the domain of FI, and principal limitations inherent to FI.

- Chapter 4 describes the first contribution of this dissertation: The design and implementation of the FAIL* FI tool. Addressing major shortcomings in the tool landscape identified in the previous section and Chapter 3, FAIL* supports complete fault-space exploration, susceptibility analyses on several levels of granularity, and the comparison of different programs. I describe the design decisions making these properties possible, and provide details on FAIL*'s implementation and extension points. Additionally, Chapter 4 presents case studies with FAIL*, highlighting analysis methods at different levels of granularity that operate on the results from complete fault-space exploration.

- Chapter 5 addresses the second contribution: A heuristical fault-space pruning technique reducing the computing efforts for complete fault-space exploration by freely trading the number of FI results for result accuracy. After explaining the basic idea of *fault similarity*, I give details of the pruning technique's implementation as an extension to FAIL*, and evaluate the approach on a set of benchmarks.

- Chapter 6 provides a complementary approach to reduce FI campaign runtimes by describing the third contribution: A technique I call *Smart-Hopping* that increases the speed of the fast-forwarding operation, which is essential for systematic, fault-space exploring FI experiments on real prototype hardware. After discussing conventional approaches for fast-forwarding on real hardware, the chapter describes the Smart-Hopping technique and its implementation in FAIL*. Its evaluation with a similar set of benchmarks shows an improvement of up to several orders of magnitude compared to the state of the art.

- Chapter 7 describes the fourth contribution: A metric for the *comparison* of the soft-error susceptibility of programs. After analyzing ill-advised practices in the field of SIHFT and uncovering the unfitness of metrics (introduced in Chapter 3) widely used to compare the fault susceptibility of different programs, I construct a new metric usable for this purpose, and provide the mathematical foundation supporting this proposition.

Chapter 8 concludes the dissertation, summarizing key results and current limitations, and suggests opportunities for future research.

1.4 AUTHOR'S CONTRIBUTION TO THIS DISSERTATION

According to §10(2) of the "Promotionsordung der Fakultät für Informatik der Technischen Universität Dortmund vom 29. August 2011", a doctoral dissertation has to contain a separate list that highlights the author's contributions to research results obtained in cooperation with other researchers. Therefore, the following overview lists the contribution of the author on the presented results for each chapter:

- Chapter 3 provides an overview on FI techniques, tools, and gaps in the state of the art. The FI-tools and techniques summaries are partially adapted from publications at PRDC 2011 [SHK+11], ARCS 2012 [SHK+12], ETS 2014 [SRS14], SAFECOMP 2014 [SBS14], DSN 2015 [SBS15], and EDCC 2015 [SHD+15]. The discussion on FI-tool generalists and specialists was already published at PRDC 2011 [SHK+11] and ARCS 2012 [SHK+12]. FI optimizations were already discussed at ETS 2014 [SRS14], SAFECOMP 2014 [SBS14], DSN 2015 [SBS15], and EDCC 2015 [SHD+15]. In each of these publications, I was the principal author, and contributed the related-work discussions adapted in Chapter 3.

- Chapter 4 describes the design and implementation of the FAIL* FI tool, and demonstrates its capabilities in four case studies. Details on FAIL* were already published at ARCS 2012 [SHK+12] and EDCC 2015 [SHD+15], where I was the principal author and provided the majority of FAIL*'s concepts, design, and implementation. The RAMpage case study was published at PRDC 2011 [SNK+11] and in IJCCBS [SKSE13], where I was the principal author, conducted all FI-related evaluation steps, and coordinated RAMpage's implementation reworking and efficiency evaluation by Ingo Korb. RAMpage's initial prototype implementation was a result of Jens Neuhalfen's diploma thesis [Neu10]. The *Generic Object Protection* (GOP) case study was published at DSN 2013 [BSS13a] and in IEEE TDSC [BSS15], which I co-authored. I contributed the FI setup with FAIL*, and the FI-related evaluation results and simulator-runtime measurements. The *Return-Address Protection* (RAP) was published at SOBRES 2013 [BSS13b], which I co-authored. Again, I contributed the FI setup with FAIL*, and the FI-related evaluation results and simulator-runtime measurements.

- Chapter 5 describes the *Fault-Similarity Pruning* (FSP) approach for FI-experiment count reduction. The approach was published at SAFECOMP 2014 [SBS14]. I was the principal author and contributed concepts, design, implementation, and evaluation of the approach.

- Chapter 6 describes the *smart-hopping* approach for fast-forwarding FI workloads in a high-latency execution environment. The approach was published at ETS 2014 [SRS14]. I was the principal author and contributed the concepts behind the *smart-hopping* approach and its checkpointing extension, the related-work dissection including the identification of the *simple-hopping* approach we used as the comparison baseline, details of the smart-hopping algorithm, and the evaluation setup. The implementation was contributed by Lars Rademacher in his master's thesis [Rad13].

- Chapter 7 dissects current practices in simulation-based FI, identifies common pitfalls regarding result interpretation, and constructs a result metric that can be used to compare hardened programs. The results were published at DSN 2015 [SBS15], where I was the principal author. I contributed the study design, the related-work dissection and resulting pitfall identification, the thought experiment, and the derivation and formalization of the *absolute failure count* metric. The identification of the underlying issues with the *fault-coverage factor* metric, and the foundations for the presented, new comparison metric were developed in cooperation with Christoph Borchert.

2 HARDWARE FAULTS AND FAULT-TOLERANT COMPUTING

"The cheapest, fastest, and most reliable components are those that aren't there."

— Gordon Bell

Contents

This chapter provides more background on the context of this dissertation by shedding light on the causes and effects of hardware faults. It subsequently defines basic terms and metrics in the context of fault tolerance, and describes possible countermeasures to hardware faults on both the hardware and software levels.

Throughout this chapter, I will focus on transient faults, also known as *soft errors*, as they are the primary focus of this work, and only scratch the surface on other fault types and their effects. Similarly, the basic principles of *software*-implemented hardware fault tolerance (SIHFT) will be described more thoroughly than their hardware-implemented pendants.

This chapter is partially based on knowledge from Shubu Mukherjee's book *Architecture Design for Soft Errors* [Muk08] and Olga Goloubeva et al.'s *Software Implemented Hardware Fault Tolerance* [GRSRV06], and complements the information with newer developments in many places where appropriate.

2.1 MOORE'S LAW AND THE EVOLUTION OF SEMICONDUCTOR TECHNOLOGY

As the introduction already outlined, the continuing exponential growth in the number of transistors per semiconductor chip – popularly known as Moore's Law [Sch97] – is accompanied by limits in chip-area sizes and power consumption. The industry's primary approach to accommodate these constraints is a permanent, iterative improvement process pushing lithography – the fabrication technology used to "print" circuits onto silicon wafers – to higher structure densities. Moore's law, of course, is no "law" in the original meaning. It was an *observation* of a trend to continuously add more transistors to integrated circuits, and a – nevertheless quite accurate – estimation of the pace of this process. Among the real intentions behind structure scaling are lower per-transistor and chip prizes [Sch97], higher switching speeds, and reduced power consumption.

Figure 2.1 shows the evolution of a specific semiconductor structure-size metric – the so-called Dynamic Random-Access Memory (DRAM) *half-pitch size*, representing half the distance between identical features in DRAM memory cells – over the last 45 years. The plot gives an impression of the steadiness of this exponential evolution, in spite of ever-repeating announcements of Moore's law coming to an end due to emerging obstacles that were assumed indomitable [Sch97].

Two examples for obstacles thought to bring Moore's prediction to an end are the "memory wall" and "power wall" problems [BJS07, Muk08]: The former was the observation that off-chip DRAM memories continuously slowed down compared to microprocessors over the years, the latter that the extreme rise in the power dissipation led to growing overheating problems. These obstacles were countered by architectural solutions, such as prefetching and multi-threading, advanced process technologies, and power-management solutions like reduced supply voltages or clock frequencies [Muk08, Mit14].

The literature expects – and already observes – that the "soft-error wall" [Muk08] or "reliability wall" [BJS07] is the next challenge in the line of obstacles on the way following Moore's law: The structure-size shrinking causes a rise in hardware-fault rates affecting the functionality of semiconductor devices. The "power wall" even exacerbates this problem: On the one hand, lower supply voltages increase the susceptibility to radiation-induced faults. On the other hand, fault-tolerance mechanisms such as *Triple-Modular Redundancy* (TMR) of critical hardware structures (see Section 2.4.2.3) consume additional power, and may be impossible to implement within given power and performance constraints [CMR15].

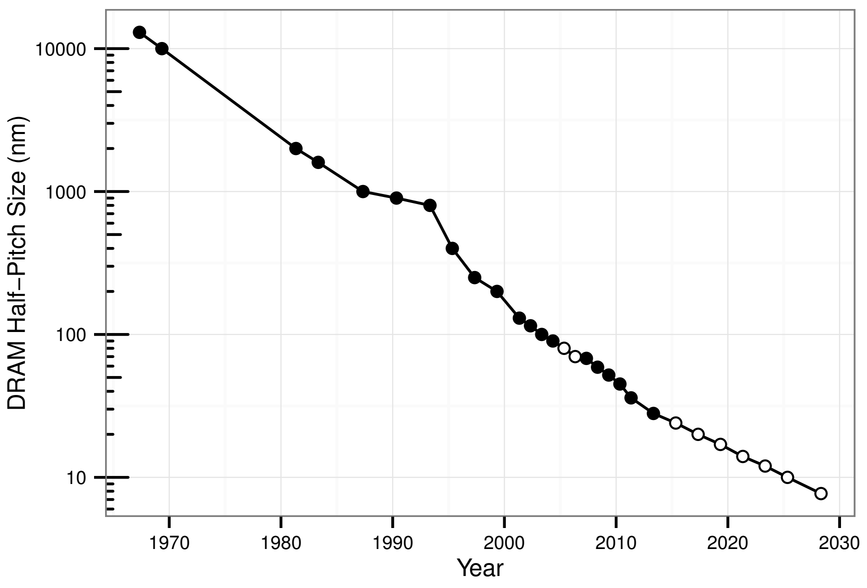


Figure 2.1: The chip industry's history and projection of the development of a central structure-size metric: The DRAM half-pitch size (log scale), representing half the distance between identical features in DRAM memory cells. [Kur05, Int08, Int10, Int11, Int13]

2.2 DEPENDABLE SYSTEMS: ATTRIBUTES, THREATS, AND MEANS

Before describing the cause and effects of hardware faults more in detail, I will first introduce basic terms and definitions from the fault-tolerance domain, used in the literature and throughout this dissertation. Some of these terms were already introduced and used intuitively in Chapter 1, but are defined more precisely here.

2.2.1 *Dependable Systems*

As defined by Avižienis et al., *"a* system *is an entity that interacts with other entities, i.e., other systems, including hardware, software, humans, and the physical world with its natural phenomena"* [ALRL04] – its *environment.* A system has a specified *functionality*, exhibits *behavior* to fulfill that functional specification, has a *state* at any point in time, and may (recursively) be composed of subsystems or *components.* The *service* a system delivers *"is its behavior as perceived by its user [... which is] another system that receives service"* [ALRL04] from the provider at its *interface.*

Dependability integrates several attributes for *dependable systems* [ALRL04]:

AVAILABILITY is generally defined by the system's *"readiness for correct service"* [ALRL04]. The system is not ready during scheduled or unplanned downtimes.

RELIABILITY is the *"continuity of correct service"* [ALRL04]. This means that that the system correctly answers to requests within the specified response time – it does not fail. [ALR04]

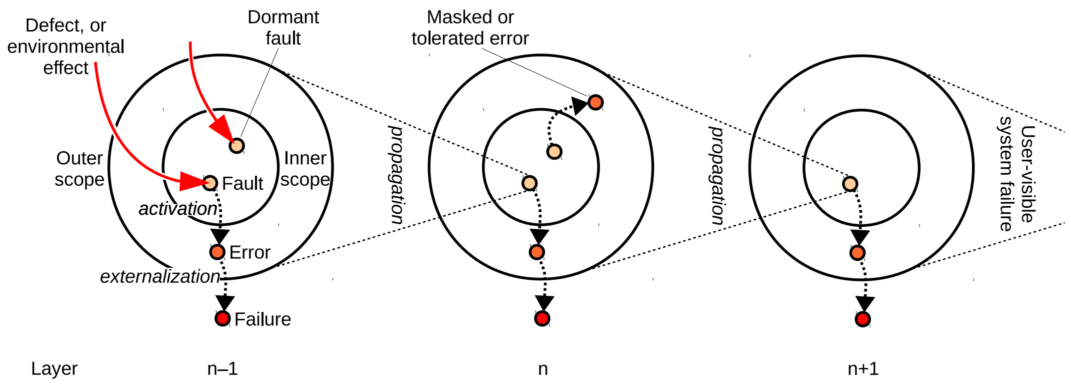


Figure 2.2: The chain of dependability threats [ALR01, ALRL04]: A *fault* may be *activated*, and turns into an *error*. An error can be externalized and turns into a *failure*, which propagates upwards as a fault on the next layer. *Figure based on [Ulb14, Muk08, Döb14].*

SAFETY describes the *"absence of catastrophic consequences"* [ALRL04] on both users and environment. This means that a passive ("fail-stop") behavior can still be "safe",[1] even if the system does not do anything useful anymore [Ech90, Muk08] – it may fail, but not in a particularly catastrophic way. [ALR04]

INTEGRITY is defined by the *"absence of improper system [state] alterations"* [ALRL04].

MAINTAINABILITY is the system's *"ability to undergo modifications and repairs"* [ALRL04].

This dissertation focuses on the dependability attribute *reliability* and its assessment. Nevertheless, reliability directly affects both *availability* and *safety*, and depends on the system's *integrity*. Maintainability is generally outside the scope of this dissertation, although I will discuss the *maintainability of FI tools* in Chapter 3 and 4.

2.2.2 *Dependability Threats: Faults, Errors, and Failures*

A complex hardware/software system can be understood as a stack of abstraction layers, each only observing the external state of the layer directly below. In this setting, the terms *fault*, *error*, and *failure* – dependability *impairments* [Lap85, LAK92] or *threats* [ALR04, ALRL04] – can be explained and put into context.

Figure 2.2 shows how a defect or an interaction with the external environment – such as the impact of energetic particles – can cause a *fault* in a

1 In his classic 1985 paper, Gray calls this *"not doing the wrong thing"* [Gra85] behavior *Reliability* instead of Safety [ALR04], and *"doing the right thing within the specified response time"* [Gra85] *Availability* instead of Reliability [ALR04]. Since then, the community has put considerable efforts into agreeing upon a common taxonomy and clear definitions of the terms used [Lap85, LAK92, ALR01, ALRL04, ALR04].

hardware structure or software module. An example on the circuit level is an energetic particle hitting a transistor. A fault does not necessarily mean that the device – as seen on abstraction layer n – has an (internal) error yet: Unless the fault gets *activated*, it stays *dormant* or *passive*. [LAK92, ALR01, ALRL04, GRSRV06]

A fault is activated and becomes an *error* if it affects the layer's internal state, residing in its *inner scope* (Figure 2.2). If, in the example, the transistor is hit "strong enough", it may change its switching behavior.

Section 2.3 describes hardware faults at the transistor level in detail.

If the layer's internal state is modified due to an error, it may not be able to provide its service to a layer above correctly – the error escalates to the *outer scope*. This externally visible deviation from the expected service is called a *failure*.[2] In the example, the transistor may be part of a DRAM circuit: The state it helped holding could be inverted – a *bit flip* occurs, constituting a *failure* on that layer.

Section 2.4 gives an overview on fault-tolerance mechanisms explicitly preventing errors from becoming failures.

Instead of turning into a failure, errors can also be *masked* – for example, overwritten before they are read – or *tolerated* – for example, due to a redundancy mechanism bringing the flipped bit back to its correct state before escalation.

Once a service failure on layer n occurred, the failure can *propagate* to the layer $n + 1$ above. From the perspective of layer $n + 1$, the layer n failure is a *fault*, possibly again being activated and turning into an error on layer $n + 1$. Hence, the *"notion of an error [...] is fundamentally tied to the notion of a* scope*"* [Muk08]. A *fault-tolerance mechanism* (see Section 2.4) on layer n can prevent a fault from propagating to layer $n + 1$ by preventing the corresponding error to turn into a failure on layer n.

Faults can further be categorized by their *lifetime*. *Transient faults* – also known as *soft faults* – appear and disappear again. *Intermittent faults* first behave similarly, but then reappear: They are considered *"early indicators of impending permanent faults"* [Muk08]. These faults, as their name already indicates, remain for an indefinite period of time, and can often only be corrected by replacing – or repairing – the affected component.

2.2.3 *Means to Achieve Dependability*

The literature categorizes *means* to actually achieve the different dependability attributes from Section 2.2.1 into four categories [ALRL04]:

FAULT PREVENTION (or FAULT AVOIDANCE) aims at *preventing* the occurrence of faults. This can, for example, be achieved by improving

2 Parts of the literature [Lap85, LAK92, Muk08] only call service deviations *visible to the user* a *failure*, and use the term *error* for all other layers. As this can lead to confusion when and how exactly an error is externalized to the upper abstraction layers, I use the term *failure* for the externalization of errors on *any* of the system's layers or subsystems (following a majority of other works, e.g., [ALR01, ALRL04, ALR04, Döb14, Ulb14]). Prasad et al. [PMW96] discuss similar dependability terminology disagreements in the community. Note, though, that the literature often uses the term *soft error* for a failure on the hardware layer, meaning a transient *fault* from the perspective of a software layer.

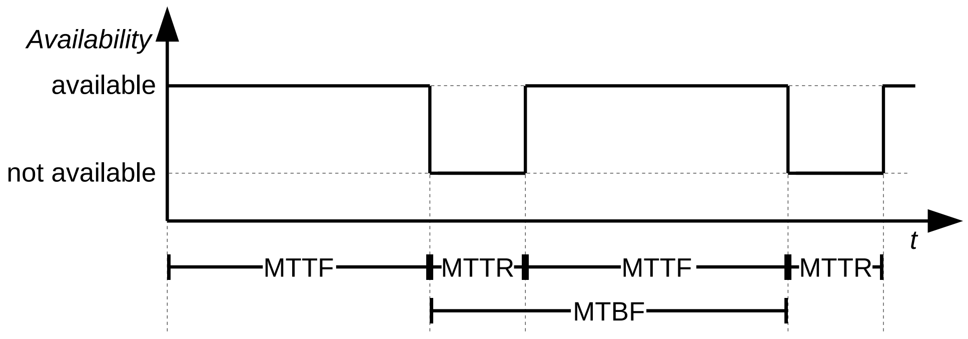


Figure 2.3: The relationship between MTTF, MTTR, and MTBF. *Figure based on [Mar11].*

chip manufacturing processes towards using packaging materials emitting less alpha radiation [Con03].

FAULT TOLERANCE is a means to "*avoid service failures* in the presence of faults" [ALRL04]. Example techniques can be found on both the hardware and software level – for example, ECC memory respectively duplication of important variables, – and include introducing some sort of redundancy into the system (see Section 2.4).

FAULT REMOVAL reduces the number and severity of faults. Fault removal can be applied at development time, for example by localizing insufficiently fault-tolerant parts of the system, and hardening them. At runtime, fault removal is achieved by conducting maintenance work, such as replacing physically damaged parts, or applying patches [ALRL04].

FAULT FORECASTING estimates the present number, the future incidence, and the expected consequences of faults. This kind of (either qualitative or quantitative) system evaluation can, for example, involve FI experiments.

In this dissertation, I focus on *fault forecasting* – the assessment and quantification of the *reliability* attribute (Section 2.2.1) – and techniques to provide information aiding *fault removal* by directed application of *fault tolerance* mechanisms.

2.2.4 *Dependability Metrics*

The literature defines several metrics to capture and quantify particular dependability attributes. One of the most basic metrics is *Time To Failure* (TTF), expressing fault or error rates. If in a specific device an error occurs after n years, this is its TTF. A more practical metric is derived when the TTFs of multiple devices – or multiple errors in the same device – are measured and averaged, yielding the *Mean Time To Failure* (MTTF). [Muk08]

Directly related to MTTF are *Mean Time To Repair* (MTTR) and *Mean Time Between Failures* (MTBF), as illustrated in Figure 2.3. MTTR measures the mean time to repair the device after the error was detected. MTBF accounts for the mean total time between one error and the next error, including the repair time: MTBF = MTTF + MTTR.

A metric related to MTTF is *Failure In Time* (FIT) [JED01], where 1 FIT *"represents an error in a billion (10^9) hours"* [Muk08]. A device's error rate is often referred to as its *FIT rate*. An advantage over MTTF is the fact that FIT rates are additive when composing a device from multiple components under specific assumptions[3] [Muk08, Mar11]:

$$\mathrm{FIT}_{\mathrm{device}} = \sum_{i=0}^{n} \mathrm{FIT}_{\mathrm{component}\,i}$$

FIT rate and MTTF[4] are reciprocally related [KK07, Muk08, Sor09, Mar11] (again, under the aforementioned assumptions):

$$\mathrm{MTTF} = \frac{10^9 h}{\mathrm{FIT\ rate}}$$

The dependability attribute *availability* (see Section 2.2.1) is directly associated with a metric quantifying it as *"the probability that a system is functioning correctly at a particular instant of time"* [Muk08]. It can be calculated as a fraction of the device's uptime and its total lifetime, or by using MTTF and MTBF:

$$\begin{aligned}\text{Availability} &= \frac{\text{system uptime}}{\text{system uptime} + \text{system downtime}} = \\ &= \frac{\mathrm{MTTF}}{\mathrm{MTTF} + \mathrm{MTTR}} = \frac{\mathrm{MTTF}}{\mathrm{MTBF}}\end{aligned}$$

In this context, the term "five nines" – or any other number of "nines" – is often used to refer to an availability of 0.99999, or 99.999 percent.

Similarly, the *reliability* $R(t)$ of a system is defined as *"the probability that the system does not experience a user-visible error* [i.e., a failure] *in the time interval* $(0, t]$*"* [Muk08], or, in other words, the probability of it still functioning at time t. Considering a population sized N of similar devices, of which at time t exactly $F(t)$ have failed, the reliability can be calculated as

$$R(t) = 1 - P(\text{Failure}) = 1 - \frac{F(t)}{N}.$$

For devices with a constant and known error rate λ, $R(t)$ is an exponential function [Muk08]:

$$R(t) = e^{-\lambda t}$$

3 Notably, the errors must occur independently and at a constant rate – a valid approximation *"during the normal operating life"* between the high-failure-rate infant mortality and wearout phases of the "bathtub" curve [Mar11].

4 The literature is actually ambiguous on whether MTTF [KK07, Muk08, Sor09, Mar11] or MTBF [Fec09, Nic10, Döb14] is the inverse of FIT. Mukherjee even contradicts himself [MER05, Muk08] throughout his publications [MEFR04]. As the definition from Marwedel [Mar11] seems more stringent, I present it here. Nevertheless, typically MTTR is much smaller than MTTF, or device repair is simply not considered at all [Muk08, Mar11]; hence MTBF ≈ MTTF, explaining the mixup in the literature [Mar11], and rendering the difference negligible in practice.

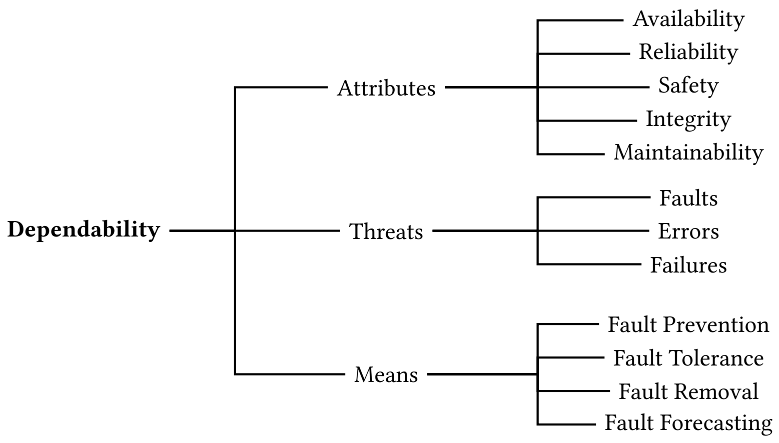


Figure 2.4: The dependability tree. *Figure based on [ALRL04].*

Evidently, the longer a device runs with a constant error rate, the less probable it becomes that it has not experienced an error. The longer a fleet of similar devices runs, the lesser fraction of devices still functions correctly – assuming that all errors invariably turn into failures.

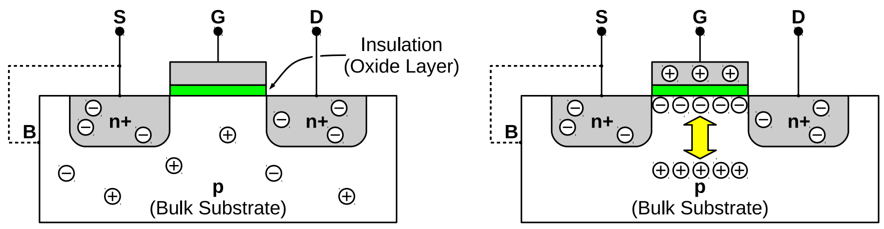
2.2.5 *Summary*

To summarize, the community has agreed on several attributes accompanying dependable systems, *reliability* being only one of them. The dependability threats *fault*, *error*, and *failure* are defined as an escalating *"sequence of events that lead to unwanted state of affairs"* [PMW96], where the three terms can be distinguished by their level of visibility to the system or its surroundings. Additionally, the means to achieve dependability are categorized into four categories, with a focus of this dissertation on *fault forecasting*. Figure 2.4 summarizes dependability attributes, threats, and means in the *dependability tree* [ALRL04]. Several metrics quantify dependability attributes, and are widely in use in the literature.

Having defined these basic terms and metrics, the next section will look into concrete instances of hardware faults at the transistor level.

2.3 HARDWARE FAULTS AT THE TRANSISTOR LEVEL

Modern integrated circuits are clearly dominated by transistors, and consequently, semiconductor structure shrinking primarily affects the dimensions and physics of transistors. To better understand the causes for hardware faults on the transistor level, I will first give an overview on how the nowadays most widely used *Metal-Oxide-Semiconductor Field-Effect Transistor* (MOSFET) works, and then describe environmental effects causing it to malfunction.



(a) MOSFET in open (non-conducting) state. (b) MOSFET in conducting state.

Figure 2.5: Model of an n-channel enhancement-mode MOSFET with source (S), drain (D), gate (G), and body (B) terminals, in open (non-conducting) and conducting states. *Figures based on [WH11, Döb14].*

2.3.1 *MOSFETs and CMOS Circuitry*

The MOSFET[5] has four terminals – *source* (S), *drain* (D), *gate* (G), and *body* (B). Usually, body and source are connected, forming a three-terminal device. Figure 2.5a shows the model of an *n-channel enhancement mode* MOSFET, or nMOS: Source and drain are connected to semiconductor material altered in the production process to hold *more* charge carriers – in this case negative charges, or electrons – than without the alteration. This production process is called *doping*, introducing impurities into the silicon semiconductor, and in this case resulting in n-type ("negative") semiconductor material. [CH02, WH11]

The two n-type source and drain areas are separated by p-type bulk substrate, which is in contrast positively charged, and therefore deprived of electrons. The boundaries between n- and p-type semiconductors act as diodes, and in effect prevent electrons flowing[6] from source to drain – or, inversely, a current flowing from drain to source – if a voltage is applied between these terminals. [WH11]

To get the MOSFET into conducting state, the *gate* electrode comes into play. It is located above the bulk substrate, and electrically insulated from it by a thin isolation layer – usually made from silicon dioxide, SiO_2. When a positive voltage between gate and body – the latter usually being connected to the source – is applied, electrons in the bulk substrate are attracted towards the gate insulator (Figure 2.5b). When the voltage crosses a threshold, the number of electrons below the oxide layer suffices to create a conducting channel between source and drain. This effect is called the *field effect*, the charge at the insulation layer is termed the *critical charge* Q_{crit} [WH11, Döb14]. The negatively charged channel is called the *n-channel*, hence the name n-channel MOSFET. In contrast, a *p-channel* MOSFET forms a *positively* charged p-channel below the oxide layer if a *negative* voltage is applied between gate and source [CH02, WH11].

5 A special case of Metal-Insulator-Semiconductor Field-Effect Transistor (MISFET) or Insulated-Gate Field-Effect Transistor (IGFET).

6 In *depletion mode*, in contrast to the described *enhancement mode*, the behavior defaults to conductibility instead.

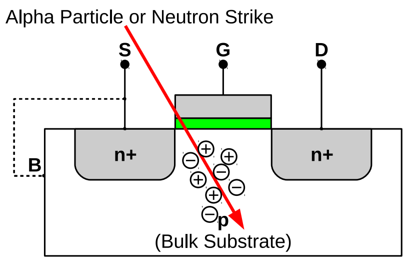


Figure 2.6: MOSFET struck by an alpha particle or a neutron, creating a trail of electron–hole pairs in the bulk substrate.

In *Complementary Metal-Oxide-Semiconductor* (CMOS) technology, which is widely in use for producing microprocessors or *Static Random-Access Memory* (SRAM), a “complementary symmetry” is achieved by symmetrically pairing up n-type and p-type MOSFETs with the effect of high noise immunity and low static power consumption. For example, one six-transistor SRAM cell – holding a single bit of data – consists of two nMOS and two pMOS transistors, forming two cross-coupled inverter gates storing the bit, and two additional nMOS transistors controlling read and write accesses. Similarly, pairs of MOSFETs are the building blocks of microprocessor logic [WH11]. In DRAM cells, in contrast to SRAM, the information is stored in a (semiconductor-based) capacitor, accessed by a single nMOS transistor [JNW08, WH11].

To summarize, being the central building block for both memory bits and CPU logic, the MOSFET’s correct functioning is vital for correct operation of a computer. For example, the malfunction of a transistor that is part of a memory cell can lead to the stored bit being flipped from 0 to 1, or vice versa. Unfortunately, as the “reliability wall” approaches, MOSFETs exhibit growing probabilities to malfunction. In the following subsections, I will summarize the physical causes for both transient and permanent faults that can lead to a MOSFET failure.

2.3.2 *Radiation-Induced Transient Faults*

This section reports on both historic and contemporary evidence for radiation-induced transient faults in semiconductor hardware, and subsequently analyzes the root causes and effects of these faults in detail.

2.3.2.1 *Evidence of Transient Faults*

Among the first reports of transistor anomalies in the literature is a 1975 study by Binder, Smith and Holman [BSH75] who describe malfunctions aboard a communication satellite in orbit, and find a likely cause in high-energy cosmic rays – especially iron nuclei. They observe a Soft-Error Rate (SER) of $1.5{\times}10^{-3}$ per transistor per year, or about 171 FIT (see Section 2.2.4). Soon thereafter, Ziegler and Lanford [ZL79] predicted the occurrence of soft errors originating from cosmic radiation *also on the ground* – calculating a

SER of about 7000 FIT for a 64 kbit DRAM chip, or about 0.11 FIT per bit – and at aircraft altitudes. Their results were treated with skepticism, until their prediction of an error increase with altitude was confirmed by data gathered in IBM's computer repair logs [ORT$^+$96, ZCM$^+$96, Muk08]. In the same year, Intel researchers May and Woods [MW79] traced transient errors in their 2107-series 16-kb DRAM chips back to a different source of ionizing radiation: uranium contamination in the ceramic packaging surrounding the chip and radiating alpha particles [MW79, Muk08]. Similarly, IBM observed radioactive contamination in an acid used for chip manufacturing [ZCM$^+$96, Muk08].

After the first observations and predictions in the 1970s and 1980s [BSH75, ZL79], many more anecdotes and systematic studies of radiation-induced soft errors have appeared in the literature. In 1995, Baumann et al. [BHS$^+$95] identified boron-10 isotopes from the manufacturing process, activated by low-energy atmospheric neutrons, as another source of ionizing radiation. Soon thereafter, Normand [Nor96] reported incidents of cosmic-ray strikes by studying error logs of several large computer systems. By the end of the millennium, Sun Microsystems observed soft errors in the insufficiently protected cache SRAMs of their UltraSPARC-II CPUs [Eva03, CCA05, Muk08].

After around the year 2000, the literature gradually moved from anecdotic, qualitative reports towards systematic, quantitative, and large-scale analyses of different technology generations. Constantinescu et al. [Con03] studied ECC errors on 193 servers, and concluded that the failure probability rises as chip densities and frequencies increase. Soon thereafter, Cypress Semiconductor reported a number of incidents arising due to soft errors [Muk08], and Michalak and colleagues [MHH$^+$05] found that some of the Hewlett-Packard many-core machines in Los Alamos National Laboratory – located at about 2200 m above sea level – crashed frequently because of cosmic ray strikes to its parity-protected cache-tag array. Li et al. [LSHC07, LHSC10] observed errors in DRAM memories in 300 machines of a web search engine and a university over a period of several months. They concluded that the observed SER of about 0.56 [LSHC07] and 0.061 FIT per Mbit [LHSC10] was lower than expected from earlier studies, which had reported between 200 and 5000 FIT per Mbit. Panzer-Steindel [Pan07] conducted a similar analysis with CERN's computers, showing that the probability of SDC errors is several orders of magnitude higher than expected by component failure statistics.

At a much larger scale, Schroeder et al. collected data sets on DRAM failures from Los Alamos National Laboratory [SG06, SG10] and Google's [SPW09, SPW11] server farms. They observed much higher error rates – on average 25 000 to 75 000 FIT per Mibit – than previously reported, and a dominance of permanent errors over transient ones. Analyzing data collected from about one million consumer-grade computers, Nightingale et al. [NDO11] reported DRAM errors to be far more likely than expected from an analysis of radiation effects. In the same year, Slayman [Sla11] summarized device FIT rate trends over several technology generations. He reported 10^{-4}

to 10^{-3} FIT per bit (equaling 100 to 1000 FIT per Mbit) for SRAM, 10^{-6} FIT per bit (1 FIT per Mbit) for Flash memory, 10^{-9} to 10^{-8} FIT per bit (0.001 to 0.01 FIT per Mbit) for DRAM, and estimated soft-error rates in logic *"per bit or per gate"* to be *"about 10 times lower than SRAM"* [Sla11]. More recently, Sridharan and colleagues conducted several studies on DRAM and SRAM cells in several large supercomputers [SL12, SSD$^+$13, SDB$^+$15]. They reported SERs between 0.066 FIT [SL12] and 0.044 FIT per Mbit [SSD$^+$13] for DRAM devices. Confirming Schroeder et al. [SG06, SPW09, SG10, SPW11] to a certain extent, they also found that permanent faults dominate over transient ones [SL12] but that this dominance diminishes with increasing device age [SSD$^+$13], and that multi-bit faults are becoming an increasing problem [SL12]. Maiz et al. [MHZA03], Radaelli et al. [RPWD05], and Georgakos et al. [GHO$^+$07] investigate the multi-bit fault *patterns* in SRAMs. Sridharan et al. [SL12, SDB$^+$15], as well as Hwang et al. [HSS12], also confirm that specific ECC memory protection schemes still significantly reduce the SER, while others do not suffice for contemporary DRAM technology [SDB$^+$15].[7]

2.3.2.2 *High-Energy Alpha Particles and Neutrons*

While some of the studies also mention other causes for soft errors, such as transistor variability originating from imperfect manufacturing processes, or device aging, radiation stands out as a basic cause throughout all semiconductor generations until today. Despite several direct and indirect origins, within the earth's atmosphere there are two primary sources of radiation-induced transient faults in transistors: *alpha particles* from radioactive decay in the close surroundings of the chip, and *neutrons* from the atmosphere.

The excursus on cosmic rays on page 29 provides more details on the origin of neutrons in the atmosphere.

Alpha particles are radiated from radioactive nuclei (such as uranium, radium or thorium) in a process called *alpha decay*, and consist of two protons and two neutrons (essentially a helium nucleus, ${}^{4}_{2}He^{2+}$). An alpha particle can affect a transistor by depositing a dense track of charge, and creating electron–hole pairs in the bulk substrate (Figure 2.6) in a quantity depending on the particle's initial kinetic energy [Muk08]. A delayed recombination of these electron–hole pairs can cause a current to flow, depositing a charge in the transistor. If this charge exceeds Q_{crit}, the transistor switches faultily, causing, e.g., the bit stored in a flip-flop to invert, or CPU logic to malfunction – a *transient* or *soft* fault occurs. Although epoxy or nonradioactive lead can be used to mostly shield a chip from alpha radiation, still small amounts of alpha particles can reach the chip. [Muk08]

Neutrons, on the other hand, only *indirectly* produce electron–hole pairs by colliding with atoms in the silicon. These collisions can produce a multitude of fragments, *"such as pions, protons, neutrons, deuterons, tritons, alpha particles, and other heavy nuclei, such as magnesium, oxygen and carbon"*

7 Additionally, Sridharan et al. [SDB$^+$15] criticize the metrics used in earlier studies, specifically mentioning Hwang et al. [HSS12] and Schroeder et al. [SPW11] as examples. The criticism suggests that previously measured memory-cell failure rates are off by several magnitudes.

Excursus: Cosmic Rays

The high-energy neutrons themselves, which the main text refers to as a possible indirect source for transistor failures within the earth's atmosphere, are descendants of a phenomenon termed *cosmic rays*. Having initially been a term for *"unknown energetic particles that interfered with studies of radioactive materials"* [Zie96], cosmic rays are nowadays commonly subdivided into four categories:

PRIMARY COSMIC RAYS are galactic particles that are believed to originate from supernovae, stellar flares, pulsars, and other cosmic activities. They primarily consist of protons – and lower quantities of alpha particles and heavier atomic nuclei – with energies above 1 GeV [Muk08]. Besides being an indirect source of terrestrial cosmic rays, primary and solar cosmic rays can also *directly* affect transistors aboard spacecraft, such as communication satellites.

SOLAR COSMIC RAYS have a similar particle-type composition as primary cosmic rays, but originate from our solar system's own sun, and have *"significantly less energy than galactic particles"* [Muk08]. (Some sources define solar cosmic rays as a subset of primary cosmic rays [Zie96].)

SECONDARY COSMIC RAYS are particles produced when primary and solar cosmic rays collide with atoms within the earth's atmosphere – primarily oxygen and nitrogen. The emanating shower of *cascade particles* – primarily pions, muons, neutrons, protons, electrons, and photons – is recursively involved in further collisions producing more cascades – this is mostly the case with particles with *strong interaction*, such as pions, neutrons, and protons. Secondary particles with a charge – protons and electrons – additionally lose energy by interacting with electrons in the atmosphere. A side effect of this process is less and less particles reach altitudes closer to sea level, with a peak at the "Pfotzer point" at around 15 km above sea level [Pfo36, Zie96]. The relative *neutron-flux increase* $I(H)$ for elevations above sea level can be approximated using the following equation, using an altitude H in kilometers [Muk08]:

$$I(H) = e^{\left(\frac{119.685H - 4.585H^2}{136}\right)}$$

Figure 2.7 plots $I(H)$ up to an altitude of 30 km. In New York City, the reference value of 1.0 at sea level has been measured in 2004 to be around 48 neutrons per hour and cm^2 [GGR$^+$04]. It varies slightly for different locations around the globe, and additionally depends on the current solar activity level [Muk08, JED01]. At a solar modulation level of 50 percent, the neutron flux is 9 percent above the New York value for Dortmund, Germany, according to the Neutron Flux Calculator [Wil06].

TERRESTRIAL COSMIC RAYS are particles that finally reach the earth's surface. Fewer than 1 percent are primary or solar particles, and they are *"mostly third- to seventh-generation cascade particles"* [Zie96].

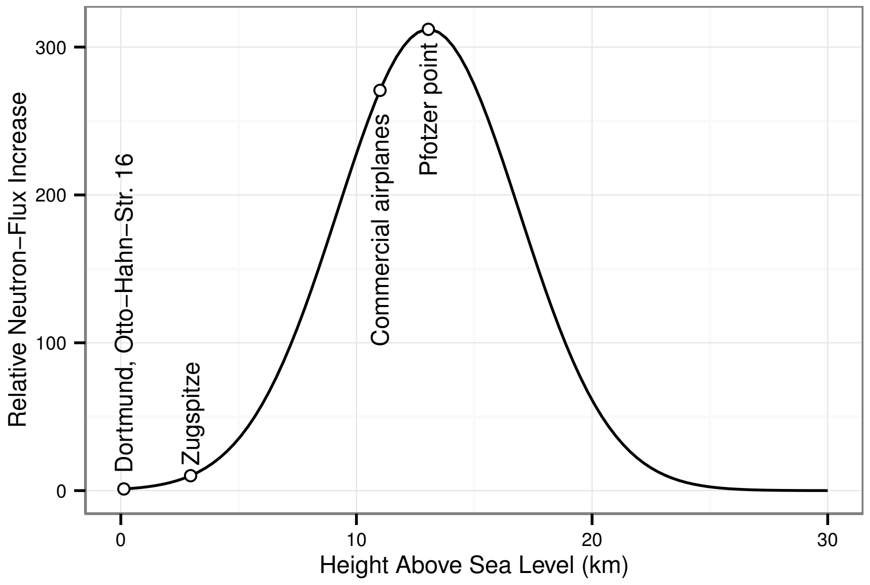


Figure 2.7: The relative neutron-flux increase $I(H)$ for elevations above sea level (see the excursus on cosmic rays on page 29): An airplane flying at a height of 11 kilometers receives 271 times more neutrons than at sea level.

[Muk08]. Subsequently, these fragments can cause ionization tracks as described for alpha particles above, and as shown in Figure 2.6.

As transistors shrink (see Section 2.1), Q_{crit} decreases, also allowing particles with less kinetic energy to affect the switching behavior. Although this might seem to *increase* the vulnerability of a single transistor to ionizing radiation, at the same time its area on the chip – and, hence, the probability of being hit at all – *decreases* [Con03]. Due to additional effects regarding manufacturing technology and the particle energy spectrum, the overall vulnerability of SRAM cells *was observed to drop* between 250 nm and 65 nm feature sizes, but started increasing again when moving to 40 nm [DW11, Döb14, Con05]. Earlier measurements indicate that the reduction of supply voltages in parallel to structure shrinking are another factor contributing to this trend change [HMA$^+$01, Con03]. As a result, the SRAM per-bit SER roughly stayed in the same ballpark across different technology generations [Bau05a, Bau05b, YE09, Sla11], but recently started to increase after years of improvement. In contrast, the DRAM per-bit SER has continuously dropped [Bau05a, Bau05b, Sla11].

Nevertheless, as the total number of transistors per chip – regardless of them being used for memory or logic – continuously grows [Sch97], the chip vulnerability also rises. In consequence, *the susceptibility to soft errors is dominated by the number of transistors used*, and increases with each chip generation. Due to the steady growth in memory density (bits per system), even DRAM can only barely hold its per-system SER throughout technology generations [Bau05a, Bau05b, Sla11].

		Faulty bit is read?			
		✗	✓		
			Bit has error protection?		
			✗	detection only	detection & correction
Program output affected?	✗	benign		false DUE	corrected
	✓	(impossible)	SDC	true DUE	corrected

Table 2.1: Classification of possible outcomes from a SBU. *Table adapted from [WEMR04, MER05, Muk08].*

2.3.2.3 *Other Sources for Transient Faults*

Although alpha and neutron radiation is assumed to be responsible for the majority of soft errors, the literature also mentions other causes, such as crosstalk of adjacent interconnects [Con03, CMV02, RCE02] resulting in line noise or signal delays. In this context, DRAM disturbance or "row-hammer" faults [KDK$^+$14] recently caught media attention, as under specific circumstances they can be triggered by software and cause security issues.

These other sources for soft errors are, however, outside the scope of this dissertation.

2.3.2.4 *Effects of Soft Errors*

Soft errors are a special case of so-called *Single-Event Effect* (SEE), which can be subdivided into further categories, describing their effect on the device they hit [Nic10, Alt13].

The classical *Single-Bit Upset* (SBU) is a radiation-induced event causing a bit to flip ("upset") in a memory cell or latch. The memory cell can be part of internal CPU state, *Instruction-Set Architecture* (ISA)-visible CPU registers, or any other part of the memory hierarchy, including CPU cache SRAM or main memory DRAM. Similar to SBUs, in a *Multiple-Cell Upset* (MCU) a single event upsets *multiple* bits, with the special case of a Multiple-Bit Upset (MBU) meaning bits in the same machine-word neighborhood. The term *Single Event Upset* (SEU) subsumes SBUs, MCUs, and MBUs, and can mean either of them. [Nic10, Alt13]

Depending on the point in time an SEE strikes, and the importance of the data bits to the device or the software that runs on it, bit-flips can result in different effects on the system level. The classification scheme in Table 2.1 distinguishes possible outcomes after a particle strike, depending on how the system copes with the bit-flip:

- If a bit flipped by an SBU is never read after the event – either because the software does not use the affected location at all, or the bit is overwritten – the system is not affected any further: The result is *benign* [WEMR04], the SBU has *no effect*.

- If the flipped bit *is* read, but it is protected by a hardware or software-implemented hardware fault tolerance mechanism (see Section 2.4), it is possible to either correct or at least detect the problem [WEMR04]:
  - If a mechanism capable of *correcting* the fault is in place, the outcome is benign. [WEMR04]
  - The case that the fault-tolerance mechanism can only detect the fault, but not correct it, results in a *Detected Unrecoverable Error* (DUE). Usually, the system reacts with a fail-stop behavior to remain in safe operation (see Section 2.2.1). The literature further distinguishes between errors that *would* have affected the program's output if not detected (resulting in a SDC, see below), and those that would have been masked afterwards (with a *benign* outcome, see below): These cases are termed *true* and *false DUE*. [WEMR04]
- If no protection scheme is in place, the flipped bit cannot be detected, let alone corrected:
  - Depending on the program, the bit-flip may be masked during further processing. This can be the case, for example, if a subsequent bit operation on the data overwrites the faulty bit, or the underlying algorithm tolerates the faulty data and still produces the correct program output. The outcome is, again, benign. [WEMR04]
  - The most unfavorable outcome sets in when the flipped bit propagates through the program until it affects the output. Avoiding such a *Silent Data Corruption* (SDC) is the foremost goal of fault-tolerance mechanisms (see Section 2.4). [WEMR04]

In the context of "soft computing" applications, such as JPEG or MPEG decoders, Thomas and Pattabiraman [TP13] further subclassify SDCs into *Egregious Data Corruptions* (EDCs) and non-EDCs. EDCs are SDCs resulting in an output that exceeds a defined fidelity threshold – for example, measured in terms of signal-to-noise ratio when compared to the correct output, – while a non-EDC output deviates in a way acceptable to the user.

To summarize, SEEs can result in outcomes with a varying degree of severity, with the SDC being the most unfavorable. Section 2.4 discusses both hardware and software mechanisms to avoid this case.

2.3.3 *Permanent Faults*

Additionally to transient – or soft – faults, which vanish after the directly or indirectly affected flip-flops or bits are re-written, integrated circuits also suffer from faults that permanently let the device malfunction.

Mukherjee [Muk08] distinguishes *extrinsic* and *intrinsic* permanent faults. Extrinsic faults originate in the manufacturing process and are usually detected by burn-in tests before shipment. One example is variability in the bulk dopant atom distribution that can lead to very low or very high threshold voltages in particular transistors, which effectively prevents them from switching [Döb14]. Extrinsic faults are an accepted part of semiconductor engineering and scaling processes, and per definition not to be expected to occur in the field.

Intrinsic permanent faults, however, are a result of material wearout due to aging and temperature stress during a semiconductor device's regular operation [Muk08]. On a chip in the field, a wearout of either the metal lines connecting transistors and capacitors, or the gate-oxide layer in transistors, can cause these faults.

One prominent metal-line failure mode is *electromigration*. In essence, electrons moving through these lines can push metal atoms forward in the electron-flow direction, and create *void* (depleted from atoms) and *extrusion* (accumulation) regions. If a *void* grows too big, the electron flow can be disrupted; an *extrusion* in the wrong place can create a new connection, and possibly a short-circuit. [Muk08, Fec09, Döb14]

The – nowadays only a few atom layers thin – gate-oxide insulation (see Section 2.3.1) can fail due to several different mechanisms. A generic wearout effect is caused by energetic particles, breaking up inter-molecule bonds in the oxide. This gradually affects the transistor's switching threshold, and hence its switching capability, up to the point of being stuck in either open or conducting state. On the other hand, *Hot Carrier Injection* (HCI) degradation and the *Negative Bias Temperature Instability* (NBTI) reduce the maximum switching *frequency*.

HCI occurs when electrons in the conducting channel of the transistor hit silicon atoms near the drain-substrate interface, producing electron–hole pairs in the drain. Some of these electrons enter the substrate, and at a sufficient energy (therefore called "hot" carriers) may damage the oxide layer. In consequence, the threshold voltage of n-type MOSFETs is increased, decreasing its switching speed. In contrast, NBTI affects p-type transistors: Under the stress of high temperatures, *"highly energetic holes bombard the channel–oxide interface"* [Muk08]. Chemical reactions along this interface permanently reduce the mobility of holes, also shifting the threshold voltage and reducing the switching speed [Muk08].[8]

2.3.4 *Summary*

To summarize, as a side effect of semiconductor device shrinking, hardware fault rates on the transistor level are on the rise. One observation is that transient – or *soft* – faults are way more frequent than intermittent or permanent faults:

8 Unfortunately, Karimi et al. [KKW$^+$15] recently showed that a malicious attacker can use specially crafted instruction sequences to even explicitly trigger NBTI, and to thereby drastically accelerate a chip's aging process.

> *"Soft errors have become a huge concern in advanced computer chips because, uncorrected, they produce a failure rate that is higher than all the other reliability mechanisms combined!"* [Bau05a]

A soft error can manifest as one or more bit-flips in the internal CPU state, in registers, in the cache hierarchy, or in main memory. These bit-flips can result in effects of different levels of graveness on the system level, from not being activated at all, to causing an undetected corruption of the program's output, known as an SDC.

As a consequence from observational studies on soft errors, and projections on their growing influence in the future, both the research community and the industry call for *countermeasures* that are capable of either preventing or tolerating them. The following section discusses such *fault-tolerance techniques* on both the hardware and the software level.

2.4 FAULT-TOLERANT COMPUTING

As hardware faults on the transistor level may propagate upwards until they become a – potentially catastrophic – system failure, various *fault-tolerance techniques* can be applied to prevent the escalation from error to failure (cf. Section 2.2.2). These techniques can be differentiated by the hardware or software layer they are usually implemented on, by which kind of errors they address, whether they are capable of correcting or only detecting errors, and their redundancy scope.

This section sheds some more light on the notion of *redundancy*, and provides an overview on fault-tolerance techniques with a focus on software-implemented mechanisms.

2.4.1 *Redundancy*

A fundamental principle in fault-tolerant systems is *redundancy*[9] – *"the property of having more of a resource than is minimally necessary to do the job at hand"* [KK07]. Once an error shows up, it can be detected – and probably corrected or masked – using redundant information, program code, time, or hardware.

Figure 2.8: Rock-climbing belay anchor.

Echtle [Ech90] differentiates *structural*, *functional*, *information*, and *temporal redundancy*. At the same time, he acknowledges that fault-tolerance mechanisms ostensibly relying on one specific type of redundancy also – emergent from the mechanism's implementation – exhibit the other three types to a certain degree.

9 Actually, fault tolerance by redundancy clearly existed before fault-tolerant computers did: Cars have spare wheels, bridges are supported by more pylons than necessary for the expected load, and the human body has two instances of some vital organs [Ech90]. In rock climbing, belay anchors are usually constructed from two redundantly bolted anchor points, often connected with a steel chain, as shown in Figure 2.8. Klingons are known to have eight-chambered hearts, supposedly allowing them to survive severe injuries even to this essential organ [Mem13].

STRUCTURAL REDUNDANCY extends the system by additional *components* that are not necessary for regular operations [Ech90]. This includes adding redundant copies of existing components, such as redundant CPUs, and new components, such as a majority voter in a TMR setup.

FUNCTIONAL REDUNDANCY adds redundant *functionality* that specifically addresses fault tolerance. Redundant functionality, such as the capability to calculate and check a parity bit, may be implemented using components that are also used for regular operations. Nevertheless, usually new components – and, hence, structural redundancy – are introduced [Ech90].

INFORMATION REDUNDANCY means storing additional information that exceeds the information necessary for regular operations [Ech90]. The aforementioned parity bit is an example of redundant information, as well as a simple copy of a program variable.

TEMPORAL REDUNDANCY denotes the additional time available to – or needed by – a fault-tolerant system for providing its service [Ech90]. For example, if parity-bit calculation is implemented in software, the fault-tolerant program needs a few more CPU cycles than without the parity calculations and checks.

The parity-bit example makes clear that all four redundancy types are involved in a concrete fault-tolerance mechanism. The redundant parity bit is a classic example for *information redundancy*, but its implementation requires additional functionality (*functional redundancy*, in the form of parity calculation) possibly necessitating new components (*structural redundancy*) and additional runtime (*temporal redundancy*).

Also note that the sole existence of redundancy does not imply fault tolerance: If an additional CPU is not actually used for tolerating faults, the system is not more fault-tolerant than without the redundant component. If a parity bit is not checked before using the associated data value, a bit flip goes unnoticed.

The unfortunate side-effect is the association of redundancy with cost: Additional hardware components cost money, require space, add weight to the system, and potentially need energy at runtime. Additional program code costs runtime performance, memory, and, again, more energy. Similarly, storing information redundantly ramps up the memory requirements of programs. Consequently, redundancy costs and fault-tolerance gains must be carefully weighed.

In the following sections I give an overview on fault-tolerance techniques on different implementation layers from the literature, and summarize the redundancy costs and fault-tolerance gains where possible.

2.4.2 *Hardware-Implemented*

The classic approach to fault tolerance has been to enhance devices on different hardware levels to mitigate soft errors. Ideally, fault tolerance on this layer provides complete fault transparency to the software, providing the illusion of fault-free operation.

2.4.2.1 *Transistor- and Circuit-Level Device Enhancements*

On the semiconductor manufacturing-process level, the industry has adopted techniques over the years that reduce the SER and also improve transistor performance. Mukherjee [Muk08] specifically mentions *triple-wells*, with which some of the electrons deposited by a particle strike are swept away and do not contribute to a faulty switching behavior anymore, and *Silicon-On-Insulator* (SOI), which reduces the deposited charge in the first place, in this context. Due to their positive effect on both SER and transistor performance, triple-wells and SOI are nowadays widely in use in the semiconductor industry, despite of the increased production costs [Muk08].

On the circuit level, the soft-error vulnerability can, for example, be decreased by "radiation-hardened" cells [Muk08]. This technique adds redundant transistors specifically to storage cells (SRAM cells or latches) while approximately doubling the chip area, increasing the energy consumption by 40 percent, and decreasing the SER 10- to 100-fold. Consequently, circuit-level techniques cannot be deployed device-wide but only for particularly critical spots.

2.4.2.2 *Memory-Protection Schemes*

One abstraction level higher, the logical and spatial relationship between multiple single-bit storage units, such as SRAM and DRAM cells, can be exploited to create cheaper soft-error tolerance under the assumption of rare and localized SEUs. The basic idea behind the field of *error-coding techniques* is to provide the minimum amount of information redundancy necessary for detecting – or correcting – an expected, localized number of bit flips while maintaining satisfactory runtime, area, and energy overhead.

A simple example of error coding implemented in hardware has already been introduced in Section 2.4.1: A *parity bit* [Ham50] holds redundant information on a set of n data bits by storing a *zero* if the number of 1-bits in the data bits is even, or a *one* if it is odd. By recalculating the parity and comparing the result with the stored parity bit – for example, just before using the associated data –, single-bit errors due to a previously occurred SBU can be detected, turning a potential SDC into a DUE (see Section 2.3.2.4). [Muk08]

A more sophisticated technology widely in use in high-end computing servers is ECC memory. In an ECC DRAM module, and similarly in ECC-protected CPU-cache SRAM, a fraction of the memory cells are reserved for parity information that is organized in a way enabling not only detection, but also correction of some error types. Usually, ECC memory implements a *Single-Error Correct Double-Error Detect* (SECDED) scheme, allowing to detect double-bit errors, and detect and correct[10] single-bit errors in a stored machine word. SECDED over 64-bit machine words incurs an overhead of eight additional check bits per word to achieve a *Hamming distance* of four [Ham50], increasing the total memory size by 12.5 percent. Additionally, both energy consumption and memory-access delays increase [Muk08, RTSM11]. Some memory-protection schemes can also correct multi-bit errors, and ramp up the overhead even further [Del97, YE10, RTSM11].

To summarize, ECC memory can detect or even correct bit-flips, but – especially for the multi-bit case, which seems to be on the rise [HSS12, SL12, SSD⁺13] – entails significant cost in terms of chip area, performance, and energy consumption – a price paid regardless how important the stored data actually is for safe and reliable operation.

2.4.2.3 *N-Modular Redundancy*

In contrast to errors in the memory hierarchy, hardware-implemented detection or correction of *control-flow errors* – or more in general, errors in the CPU logic – is usually implemented on a more coarse-grained (also termed *"macro-redundancy"* [RCV⁺05a]) level.

The classic *structural redundancy* solution, known especially from highly dependable systems in airplanes, is *N-Modular Redundancy* (NMR) of important components. For example, in a *Dual-Modular Redundancy* (DMR) setup with two CPUs that synchronously (in *Lockstepping* mode) execute the same code, a functionally redundant *voter* component can continuously compare the internal state or external behavior of the CPUs, and detect a deviation [HHJ90, SAIC⁺99]. In a TMR setup, a third component can be used to even *correct* one faulty instance without having to repeat previous work (Forward Error Recovery or FER).

The main disadvantages of hardware NMR setups include significantly reduced performance, the high cost for multiple copies of the replicated component and the additional voter, and the added complexity, weight, and energy consumption [Muk08]. As a consequence, TMR setups are limited to highly critical systems in avionics, such as the Boeing 777 primary flight computer [Yeh96], or aboard spacecraft, as for example the FPGAs responsible for the descent and landing phases of the Mars rovers *Discovery* and *Spirit* [Rat04].

2.4.2.4 *Hardware Multithreading and the Sphere of Replication*

On a more fine-grained level, some hardware-implemented fault-tolerance solutions make use of the redundancy available in modern CPUs for per-

10 As the corrective action repairs the flipped bit and allows the system to continue normal operation, it is often referred to as a Forward Error Recovery (FER) or Forward Error Correction (FEC) method. In contrast, Backward Error Recovery (BER) resolves the situation by restoring and restarting the system from an earlier recorded state, a *checkpoint* [Muk08]. For example, the *ReStore* architecture [WP05] implements BER by restoring a CPU-state checkpoint when an error is detected; *SafetyNet* [SMHW02] solves this problem in a lightweight but coarse-grained manner for the multiprocessor case.

formance improvements, such as pipelining, *Simultaneous Multithreading* (SMT) [TEL95], or *Chip-level Multiprocessing* (CMP). For example, Rotenberg's AR-SMT [Rot99] executes the instruction stream a second time with a small delay, using SMT resources of a single CPU core and additional hardware checker mechanisms, and reports a performance penalty of 10 to 30 percent. Reinhardt and Mukherjee's Simultaneous and Redundantly Threaded (SRT) processor [RM00] achieves better performance [RCV⁺05c] by comparing the streams only when memory accesses occur, with the SRTR variant also providing recovery [VPC02].

In this context [RM00], Reinhardt and Mukherjee also introduce the concept of the *Sphere of Replication* (SoR):

> *"Components inside the sphere of replication enjoy fault coverage due to redundant execution, while components outside the sphere do not [...]. Values that enter and exit the sphere are the inputs and outputs that require replication and comparison, respectively."* [RM00]

In the use case of the SRT processor, the SoR is exited when memory accesses occur. Consequently, hardware-thread synchronization and error detection take place during memory accesses. [RM00]

2.4.3 *Software-Implemented*

Besides the widely deployed manufacturing-process level measures, most hardware-implemented fault tolerance methods tend to be costly, and entail increased energy consumption and in many cases a performance degradation. Naturally, the redundancy necessary for fault tolerance comes at a price, but it currently only tends to be paid for expensive server systems – for example, ECC memory, or CPU Lockstepping [HHJ90, SAIC⁺99] – or particularly vulnerable and critical devices aboard airplanes [BT93, Yeh96] or spacecraft [Rat04].

In contrast, *Software-Implemented Hardware Fault Tolerance* (SIHFT) aims at improving fault tolerance without relying on specific hardware features. The envisioned cost reduction – in comparison to the hardware-implemented counterpart – stems not only from the ability to use COTS hardware, but also from the potential to take advantage of application knowledge. For example, only "critical" data structures of an application need to be enhanced with information redundancy – for example, a parity bit – and functional redundancy – such as a parity-bit calculation and check routine, – unlike ECC memory, which protects *every* data bit in memory regardless of its importance. Similarly, only the most error-prone parts of a program can be run multiple times for TMR voting, instead of triplicating the whole CPU (structural redundancy) and adding a hardware-voter component.

Additionally to the cost reduction – and, assuming an elaborate application-specific fault tolerance setup, potentially an acceptable performance and energy consumption – SIHFT also benefits from other software-specific advantages. The development cycles of software are usually much shorter than those of hardware, allowing the incremental development of fault-tolerance techniques and their application-specific tailoring. Additionally, SIHFT-protected devices can easily be updated in the field to accommodate changed fault-tolerance requirements or adapt the device to a new environment.

The disadvantages of SIHFT include the fact that implementing the *same* fault-tolerance scheme in software – ignoring the potential for application-specific tailoring – can cost orders of magnitude in performance and energy consumption when directly compared to hardware implementations. Additionally, SIHFT cannot reach architectural state that is not visible on the ISA level, such as, for example, shadow registers, or cache-line tags, and consequently cannot protect it.

In the following, I give an overview on existing SIHFT solutions including the costs they involve, and derive resulting requirements on reliability analysis.

2.4.3.1 *Inherent Fault Tolerance*

Instead of constructing explicit SIHFT solutions, a part of the literature analyzes, classifies, and improves algorithms targeting their *inherent* fault-tolerance properties. An intuitive example for inherent fault tolerance is Newton's iterative method for approximating the roots of a real-valued function: If, due to a soft error, the CPU miscalculates one approximation step, or the current approximation value is corrupted in memory, the subsequent iterations gradually correct the error without any need for explicit fault tolerance. Other examples from the literature are *Artificial Neural Networks* (ANNs), which have been shown to be intrinsically tolerant to soft errors on board of a research satellite [VCT+99], or works from the algorithms community on fault-tolerant variants of sorting and searching algorithms [FI04, FGI05, FFI06, FGI07, FI08, FFI09, FGI09]. In this context, even an own research field called *Algorithm-Based Fault Tolerance* (ABFT) has developed, producing fault-tolerant variations of classic numerical algorithms such as matrix multiplication [HA84, Tyr96] or number factorization [LCWV13].

2.4.3.2 *Leveraging Hardware Detectors*

A more generic, straightforward and widespread approach for *detecting* hardware errors – or their indirect effects – is to leverage existing detectors in modern microprocessors. These mechanisms usually issue a CPU *trap* or *exception* when they detect a division by zero, the execution of a malformed op-code, or a memory access not permitted by a *Memory-Management Unit* (MMU) or *Memory-Protection Unit* (MPU). Although the primary use of CPU exceptions, and memory isolation in general, is the detection – and escalation prevention – of programming errors and the enforcement of a security model, these concepts as well can help detect hardware errors. Consequently, many SIHFT techniques include hardware detectors in their solutions [MKGT92, AN97, ANKA99, RCV+05c, HLDL15]. For example, Miremadi et al. [MKGT92] and Hoffmann et al. [HLDL15] fill unused memory with a trap-causing instruction to detect erroneous jumps to these areas.

Additionally, most developers of SIHFT solutions that do not *explicitly* rely on hardware detectors *implicitly* use them in their evaluation: In many cases, test runs in faulty environments – or under FI (see Chapter 3) – that end up with a CPU exception are allotted to the *detected* (or DUE, see Section 2.3.2.4) category even if the solution does not explicitly mention an exception handler.

2.4.3.3 *Executable Assertions for Data Error Detection*

Executable assertions are statements or code sequences that check statically known state invariants, preconditions, or postconditions, in specifically chosen points of a program at runtime [Sai77, MAM84, RBQ96]. As Saib states, they *"can detect errors in input data and prevent error propagation"* [Sai77]. The type of assertions described in this section achieve detection *without* adding more state (information redundancy) to the program.

Primarily known for their use as detectors for software bugs[11] – and, hence, as an aid in software testing [MAM84] and debugging, – assertions are also widely used to detect *data-flow errors* caused by hardware faults [GRSRV06].

Both the assertion placement and the check conditions themselves are highly application specific, and require intimate knowledge about the target program. For example, Rabéjac et al. [RBQ96] informally define a design process with instructions on how to manually devise invariants, and pre- and postconditions on functional blocks. Hiller [Hil99, Hil00] approaches this problem more systematically by classifying internal variables into behavioral categories, and defining generic assertions detecting behavioral violations. The placement problem is not considered.

Due to the high application specificity, the reported effectiveness achieved by the different solutions varies wildly – from a tiny 0.23 percent average coverage improvement [CRA06] up to 100 percent SDC coverage for Hari's maximum configuration [HAN12]. Similarly, the static code (5.2 percent [SK08a] up to 15 percent [GŁM+09]) and dynamic runtime (almost zero [CRA06] up to 35 percent [LJ11]) overheads strongly depend on the concrete solution and the modified benchmarks.

In essence, data-error detecting executable assertions necessitate program analysis techniques that provide promising *locations for placing assertions* in the target program.

2.4.3.4 *Information Redundancy*

Although classic executable assertions can be used to detect data errors (see Section 2.4.3.3), they can only detect deviations from the set of possible – and,

11 For example, the C standard library `assert` macro is documented as a means *"to help the programmer find bugs in his program"* [Koe93], and *"generates no code"* [Koe93] in release builds of the program.

hence, *self-consistent* – system states. With additional information redundancy (see Section 2.4.1), they can also detect deviations that are reflected in *consistent but wrong* state. Consequently, some techniques cope with hardware faults, such as single-bit flips, that *directly* affect values stored in variables.

The simplest approach to implement information redundancy is the *duplication* of variables. When a value is written to a duplicated variable, it must also be written to its redundant copy; when a value is read, the duplicate is read as well, and the values can be checked for identity to detect a corruption. For example, the *REliable Code COmpiler* (RECCO) by Benso et al. [BCPT00, BDCDN+01b] implements a source-to-source transformation that duplicates a subset of the variables used in a C/C++ program, and inserts code to regularly update and check the consistency between original and *shadow* variable. Similarly, Xu et al. [XTS08] describe the *Optimum Data Duplication* (ODD) approach selecting variables based on a custom error-propagation metric [XST08].

Besides duplication, also more memory-efficient *error coding* techniques [Muk08] have been implemented in software. Although many of these techniques were originally devised to be implemented in hardware (see Section 2.4.2.2), they naturally can also be carried out in software. For example, Shirvani et al. [SSM00] analyze four software ECC implementations with different memory-scrubbing intervals, and deploy the mechanism on the ARGOS satellite [SM98, SOM+00, LWW+02]. Nicolescu et al. [NVSR01] calculate a *Cyclic Redundancy Check* (CRC) code [PB61] of machine-code sections before executing them, and compare the result with correct values calculated at compile time. Additionally, they manually transform the target program to encode variables with a Hamming code, and to detect and possibly correct errors on each variable use.

Another information-redundancy technique initially implemented in hardware – for example, in the STAR computer [AGM+71] – are *arithmetic error codes*, described by Avižienis as *"the class of error-detecting and error-correcting codes which are preserved during arithmetic operations"* [Avi71]. Arithmetic coding allows specific calculations directly on *code words* without decoding them first, and thereby maintains the data protection throughout the whole program. A widespread example are *AN codes* [Bro60] and their improved *ANB* and *ANBD* variants [For89]. The code's name "AN" directly describes the encoding operation: The program-wide constant A is multiplied to the number N – the data word – to obtain the code word. A code word is valid if the remainder after dividing by A is zero; otherwise, an error occurred. AN codes stay valid during several arithmetic operations, for example – due to distributivity – addition or subtraction:

$$AN_1 + AN_2 = A(N_1 + N_2)$$

Besides addition and subtraction, there exist AN-preserving equivalents to multiplication, division, and logical operators [Ulb14]. Software implementations of AN codes – and their ANB and ANBD variants – have been successfully used by, for example, Schiffel, Fetzer et al. [WF07, FSS09, SSSF10a, Sch11]. Hoffmann et al. [HUD+14b, HUD+14a] (work I also contributed to) thoroughly analyze practical implementation issues of AN encoding, and give concrete advice which values for A are particularly advantageous.

Depending on the choice which subset of program data to protect, information-redundancy techniques can come with a considerable runtime and data-memory overhead. The simpler data-duplication approaches range between 6 [BCPT00] and up to 173 percent [PGZ08] runtime overhead, require from 18 [BCPT00] up to 177 percent additional memory for the data copies, and are reported to achieve up to 99.7 percent fault coverage [XTS08]. On the other hand, arithmetic error codes entail magnitudes more runtime overhead, but also very good fault coverage: For example, the ANBDmem encoding by Schiffel et al. [SSSF10a] slows down the encoded application by up to a factor of 385 – or, by 38 400 percent, – but reportedly has a fault coverage of up to 99.97 percent.

Not least owed to the partially extreme costs regarding space and runtime overhead, all information-redundancy techniques require analysis techniques that provide information on the *most critical data structures* in the target program.

2.4.3.5 *Control-Flow Monitoring*

Unlike the type of executable assertions described in the previous sections, which are intended to application-specifically (Section 2.4.3.3) or -unspecifically (Section 2.4.3.4) detect *state* inconsistencies, executable assertions are also a common technical means to implement *Control-Flow Monitoring* (CFM). CFM aims at detecting *Control-Flow Errors* (CFEs) – errors that cause the CPU *"to fetch and execute an instruction different than expected"* [GRSRV06]. This can happen, for example, through a directly or indirectly corrupted instruction-pointer register – the latter, for example, caused by a bit-flip in a function pointer or return address, or a wrong jump-target calculation in the ALU, – or an inverted branch decision.

Most CFM techniques are based on the partitioning of programs into *Basic Blocks* (BBs) [All70, GRSRV06]. A program's Control-Flow Graph (CFG) is composed of BB nodes and branch edges denoting statically valid control-flow transitions [GRSRV06]. At runtime, each time reaching the end of a specific basic-block node, only *one* of the statically valid edges emanating from this node is the *right* edge – depending on the branch condition's actual parameters. From these basic terms, Goloubeva et al. [GRSRV06] define five types of CFEs:

1. A fault causes an *illegal* (not in the CFG's edge set) branch *"from the end of a BB to the beginning of another BB."* [GRSRV06]

2. A fault causes a legal but *wrong* branch (according to the branch condition, a different legal branch should have been taken) *"from the end of a BB to the beginning of another BB."* [GRSRV06]

3. A fault causes a branch from the end of a BB to an arbitrary program instruction that is not a BB start. [GRSRV06]

CFE types 4 and 5 address two flavors of spontaneous jumps or instruction-pointer changes where the originating instruction is not a branch instruction at all [GRSRV06].

In general, CFM aims at detecting a subset of these CFE types, and achieves this by introducing additional global state that records at runtime which BBs or control-flow edges were visited recently, and by inserting recording and checking code into the beginning or end of the BBs themselves. For example, Miremadi et al. [MKGT92] describe the *Block Signature Self Checking* (BSSC) scheme – and an improved *Block Entry Exit Checking* (BEEC) variant [MT95], – which protects the control flow by a block-specific signature that is initialized at the beginning and checked at the end of each BB, and a constant inter-BB state. Benso et al. [BDCDN$^+$01a] model the valid control flow with regular expressions in a separate checker process, and transform the program with a source-to-source-compiler to transmit control strings via IPC. Kanawati et al. [KNKA96] and Alkhalifa et al. [AN97, ANKA99] describe *Control-flow Checking using Assertions* (CCA) and, respectively, *Enhanced CCA* (ECCA).

Depending on the CFE-type detection capability and achieved CFE coverage, existing CFM techniques implicate program-size overheads of 5 [RBQ96] to 175 percent [YC80], and a *"negligible"* [SS94] up to 140 percent [YC80] additional runtime.

In consequence, depending on their overhead, CFM techniques must be deployed in the target program with care, and the reliability of the resulting, hardened program variants must be iteratively monitored and *measured*.

2.4.3.6 *Redundant Multithreading*

Another fault-tolerance technique adapted from hardware to software is *Redundant MultiThreading* (RMT) – temporal (or structural) redundancy by repeated execution of the same code, and detection by comparison or correction by voting. Solutions differ mainly by the replication granularity, replica comparison frequency – and, consequently, detection latency, – the coverage of their SoR (see Section 2.4.2.4), and detection versus correction capabilities [Pig05, GRSRV06].

On instruction-level granularity, one of the most-cited RMT solutions is *Error Detection by Duplicated Instructions* (EDDI) by Oh, Shirvani and McCluskey [OSM02]. EDDI duplicates every instruction using different registers and copies of variables in the duplicates, and inserts consistency checks before every memory-store and conditional-branch instruction. Thereby, the technique automatically generates a hardened variant of the program with more than twice the number of static and dynamic instructions. Although such compiler-supported instruction-level RMT solutions tend to hide some of their runtime overhead in the instruction-level parallelism of modern super-scalar CPUs, the overhead of the non-selectively applied solutions can reach up to 113 percent [OSM02]. Nevertheless, the achieved fault coverages come close to 100 percent [OSM02, RCV$^+$05b, CRA06].

Although the literature is dominated by instruction-level RMT, some approaches target other granularity levels. For example, Li and Hong [LH07b] implement RMT – and function-parameter duplication – on the statement level of the C programming language, but do not mention tool support to automate this task. Ulbrich, Hoffmann et al. [UHK$^+$12, HUD$^+$14b, HUD$^+$14a] describe *CoRed*, a task-level RMT setup with a hardened voter for a control application. Döbel et al. [DHE12, DH12, DMH14, Döb14] extend the L4/Fiasco.OC microkernel by *Romain*, an approach for transparent, application-level RMT, and later also cover the case of applications that are multithreaded themselves [DH14, Döb14].

As coarse-grained RMT solutions cannot hide their redundant execution with instruction-level parallelism, their runtime overhead usually exceeds 100 percent. As a hardware-resource remedy, many solutions assume unused CPUs in a multi-core setup that can be used to run replica without the additional wallclock-time penalty, but still require more runtime for input replication and output comparison [SMR$^+$07, SBM$^+$09].

In essence, fine- or coarse-grained RMT solutions must be applied to target programs in a dosed manner, and the reliability of the resulting, hardened program variants must be iteratively monitored and *measured*.

2.4.3.7 *Hybrid Hardware/Software Techniques*

Some SIHFT approaches require additional hardware support to improve performance, or to cover errors that are particularly hard or even impossible to detect with a pure software mechanism. The classic example is the *watchdog timer* that resets the monitored CPU if it does not receive a specific command from the software part of the fault-tolerance implementation for a predefined time period. For example, Olsson and Rimén [OR95] complement their CFM approach by a watchdog timer that detects the absence of block signatures.

2.4.3.8 *Language Support*

Orthogonally to the development of SIHFT techniques, some research explicitly concentrates on software-engineering aspects of fault tolerance – its encapsulation in software modules, and language support for its implementation.

For example, Gawkowski et al. [GRS10] choose classic modularization with their *AppHard* library, which provides functionality for checkpointing and instruction-level RMT, and formulate the goal of usage simplicity for the application programmer. Similarly, Huang and Kintala's *libft* [HK93] encapsulates checkpointing and recovery, and provides facilities for N-version programming [CA78].

As most fault-tolerance mechanisms – besides coarse-grained RMT – require large parts of the application to be modified on the machine- or source-code level, they cannot easily be encapsulated in ordinary libraries. Some approaches provide their own compilers or compiler modifications, but at the cost of little portability and a dependency on additional tools. To ameliorate these problems, some research approaches harness *reflection* [Smi82] mechanisms of specific programming languages, or alternatively *Aspect-Oriented Programming* (AOP) [KLM+97] languages, to modularize fault-tolerance implementations that would otherwise crosscut the program.

For example, Alexandersson et al. [AÖI05, AÖ07, AÖK10, AÖ10, AK11] implement various SIHFT techniques using *AspectC++* [SL07], an AOP extension to C++, which they extend by means to detect and control data accesses. Similarly, Afonso et al. [ASB+08, Afo09] implement task-level triple-modular RMT with AspectC++ in the distributed BOSS operating system, and report a good *separation of concerns*, code reusability, and the advantage of fault-tolerance *configurability*. Being AspectC++ users as well, Hameed et al. [HWS10] apply different fault-tolerance mechanisms to a robot-control application, and analyze several metrics quantifying the *separation of concerns*. Borchert et al. [BSS12, BSS13a, BSS13b, BSS15] (work I also contributed to) implement classic data-duplication and error-coding techniques in AspectC++, and partially specialize them for particular *eCos* OS components. They extend the AspectC++ language by compile-time introspection capabilities, allowing template meta-programs to statically generate efficient information-redundancy code for a configurable set of C++ classes.

All fault-tolerance techniques supported by specific language features or constructs share the requirements of the previously mentioned SIHFT techniques: They need to be deployed on the *most critical modules or data structures* of the target program, require placement or configuration guided by a previous analysis step, and the result must be *measured and compared* with the original version of the program.

2.5 SUMMARY

As semiconductor devices – and particularly transistors as their dominating component – move towards the "reliability wall" by continuous shrinking, microprocessors and memory devices are more and more threatened by radiation and aging effects. These dependability threats can cause – predominantly transient – bit-flips in the device state, possibly resulting in DUEs or SDCs that affect system attributes such as reliability, safety, and integrity.

2.5.1 *Fault Tolerance*

As countermeasures, both industry and the research community devised hardware- and software-implemented hardware-fault tolerance techniques. These techniques are invariably based on different types of redundancy to avoid or tolerate soft errors and their propagated effects.

Hardware-level techniques – besides manufacturing-process level techniques such as triple-wells and SOI – are relatively expensive, tend to significantly increase power consumption, and in some cases decrease performance. Especially for memory-protection schemes that equally provide redundancy for all bits in memory, the obliviousness to application demands creates more costs than necessary in many cases, as most parts of the memory store uncritical information. Not surprisingly, the most expensive hardware-implemented fault-tolerance techniques, particularly N-modular redundancy in its coarse-grained TMR variant, are only used in the most critical systems in avionics and spaceflight.

SIHFT techniques, on the other hand, have the potential to take advantage of application knowledge by only applying them to the most critical spots in the target system. Nevertheless, SIHFT can entail enormous overheads of up to several orders of magnitude if it is deployed system-wide without exploiting this potential. Moreover, most SIHFT techniques aim at detecting or correcting *one specific type* of error – data, control-flow, or computation errors, – which led to the development of hybrid techniques combining the error-type coverage of different software- and also hardware-implemented techniques.

2.5.2 *Reliability Analysis, Measurement, and Test*

A direct consequence of the high overhead in space and time of most SIHFT solutions is the necessity to *analyze* target systems: This *reliability analysis* step must yield the most critical spots in the program – for example, the most important variables to protect with an information-redundancy solution – to avoid a too costly system-wide hardening. Or, as Taylor et al. [TMB80] put it in their 1980 paper:

> *"A little redundancy, thoughtfully deployed and exploited, can yield significant benefits for fault tolerance; however, excessive or inappropriately applied redundancy is pointless."* [TMB80]

After analyzing the software for its most critical spots, and hardening those with the chosen SIHFT solution, the resulting system needs to be *measured* to determine the protection's effectiveness and efficiency, and to possibly compare the hardening with other SIHFT variants. Additionally, SIHFT implementations must be *tested* during their development to avoid fault propagation through a system hardened with a partially dysfunctional fault-tolerance mechanism.

Consequently, in the next chapter I discuss fault-tolerance assessment methods for analysis, measurement, and test from the literature. The discussion will focus on FI-based techniques and tools directly related to this dissertation, but also provide an overview on alternative methods.

3 FAULT TOLERANCE ASSESSMENT AND HARDWARE FAULT INJECTION

"Failure free operations require experience with failure."

— Richard I. Cook, *How Complex Systems Fail* [Coo98]

Contents

As *Software-Implemented Hardware Fault Tolerance* (SIHFT) increases in popularity, demand grows for methods to *test* its implementation, to *measure* and *compare* its effectiveness, and to find *cost-effective mechanism placements* in a concrete target-software environment that do not diminish all gains from semiconductor-device shrinking. This chapter discusses the widespread use of *Fault Injection* (FI) for these purposes, and reviews existing techniques, tools, and optimizations for FI. In this context, I focus on *simulation-based* FI, and identify gaps and problems in the state of the art. Additionally, I give an overview on alternative analysis techniques and heuristics for SIHFT placement – the identification of critical spots –, which partially but not necessarily also rely on FI results. Finally, I summarize observations and gaps in the state of the art, which will form the basis of the requirements for the Fail* FI tool in the next chapter.

Parts of this chapter were originally published on PRDC 2011 [SHK$^+$11], ARCS 2012 [SHK$^+$12], ETS 2014 [SRS14], SAFECOMP 2014 [SBS14], DSN 2015 [SBS15], and EDCC 2015 [SHD$^+$15].

3.1 FAULT INJECTION

Unlike testing *functional* software properties, testing whether a SIHFT mechanism works correctly, assessing whether the system exhibits only expected failure modes, and measuring failure rates – and, hence, reliability, – usually cannot be achieved under normal operating conditions [BF93]. Within the earth's atmosphere, FIT rates of contemporary semiconductor hardware (memories, CPUs, etc.) are so low that the software is extremely rarely exposed to faults – which even more rarely propagate and escalate to observable, system-wide failures. As Marwedel aptly puts it:

> *"It is frequently not possible to experimentally verify failure rates of complete systems. Requested failure rates are too small and failures may be unacceptable. We cannot fly* 10^5 *airplanes* 10^4 *hours each in an attempt to check if we reach a failure rate of less than* 10^{-9}*!"* [Mar11]

Usually – for the sake of testing and measuring SIHFT – only specific subsystems are examined, and consequently, failures do not mean to sacrifice whole planes. Nevertheless, too low fault and failure rates prohibit simply waiting for natural causes to trigger hardware faults[1], unless a testbed on board a satellite – with much higher failure rates in space – can be used to examine a fault-tolerance implementation [SOM$^+$00, LWW$^+$02]. And – even *if* soft errors occur, this approach does not yield statistically authoritative observation numbers that can be used for testing, measurement, or comparison.

The classic approach to solve the low fault-rate problem is the artificial acceleration of fault occurrences by *Fault Injection* (FI). FI is used to mimic the effects of the original radiation-related causes for soft errors – or, as well, permanent faults – to a certain degree, but with extremely increased occurrence probability to trigger SIHFT mechanisms often enough for sufficient evidence of their effectiveness.

1 For example, the MTTF of a 8 GiB DRAM device with 0.044 FIT per Mbit [SSD$^+$13] – i.e., the average time until *any* one of the 2^{33} stored bits flips due to a natural cause – is about 40 years. Similarly, an 8 MiB SRAM device with 10^{-3} FIT per bit [Sla11] sees its first bit flip after an average of eight months.

3.1.1 *Purposes of Fault Injection*

FI – and related analysis techniques discussed in this chapter – can be used for different but superficially very similar purposes in the context of SIHFT evaluation. As their differences play a role in the requirements on FI tools and techniques, in the context of this dissertation the following purposes must be clearly distinguished:

TESTING aims at empirically answering questions in the context whether a developed fault-tolerance mechanism – for example, an EDM or ERM – works correctly. *Does the mechanism get triggered when a hardware fault propagates as an error in a protected data structure? Does the recovery allow the system to continue normal operations? Are all failure modes of the system expected?* Testing – with varying fault models (see Section 3.1.3) – is the dominant purpose for FI in the recent literature [Giu14], and provides mostly *qualitative* insights on a system's behavior under error conditions. In the terminology introduced in Section 2.2.3, testing falls under the *fault removal* category [BP03].

MEASUREMENT adds quantitative evaluation to the testing process, such as the application of metrics like MTTF, reliability (see Section 2.2.4), or the fault-coverage factor (see Section 1.2.6 and Section 3.1.2.5) to the system. Although the used FI techniques may be similar or the same as for testing, much more care must be taken when deciding on the fault types and distribution. Measurement results can be used to determine whether the system meets specific standards, or for *comparison* against other systems or system configurations. In classical dependability terms (see Section 2.2.3), measurement belongs to *fault forecasting* [BP03].

ANALYSIS extends beyond system-wide measurement by providing *fine-grained* quantitative results on the system – for example, on a module, function, variable, or source-code line level. This information aids in localizing critical spots of the system that should be protected by SIHFT techniques. FI is only rarely used for fine-grained analysis.

The literature sometimes does not explicitly distinguish between these purposes; especially testing and measurement tend to get confused when SIHFT mechanisms are tested, but fault-coverage measurements are presented.

Before surveying actual FI techniques and tool implementations in Section 3.2, the following sections give a generic overview of the basic components involved in most FI tools, and introduce necessary terms and definitions.

3.1.2 *The FARM Model*

In their classic FI paper, Arlat et al. [AAA+90] introduce the FARM model, which subsumes FI with the attributes *faults*, *activations*, *readouts*, and *measures*. Goloubeva et al. [GRSRV06] concretize the relatively abstract original

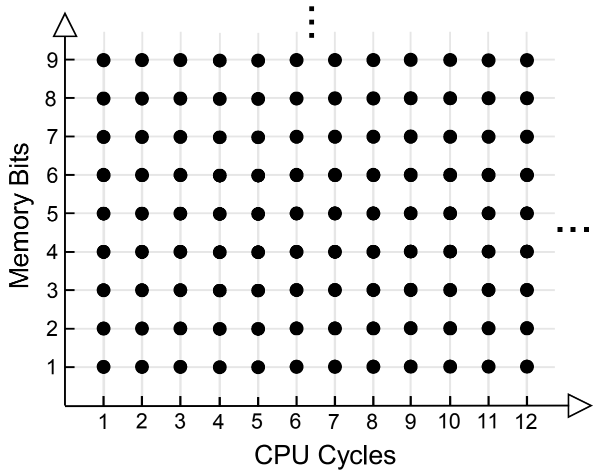


Figure 3.1: Single-bit flip fault space for a run-to-completion workload. Each dot represents a possible FI time and location coordinate. The workload proceeds until and stops at a specific point in time, the corresponding memory bit is flipped, and the workload resumes.

definitions and improve their comprehensibility, which is why I summarize their interpretation in this section.

3.1.2.1 *Fault Set, Fault Space, Campaign, and Experiment*

In the FARM model, F is *"the set of faults to be injected"* [GRSRV06] during a so-called FI *campaign*. Each fault is characterized by its coordinate in the *fault space*, which is spanned by the FI *time* and the fault *location*.[2] The FI time denotes the point in time after starting the target workload *when* the fault is injected, and is, for example, expressed in clock cycles. The fault location decides *where* the fault is injected, for example at a specific memory address and bit position. Figure 3.1 shows the two-dimensional fault space with all possible time/location pairs depicted as dots. In each single FI *experiment* of the campaign, one fault from the fault set is actually injected.

3.1.2.2 *A Fault-Injection Environment*

In their 1997 FI techniques and tools survey, Hsueh et al. [HTI97] describe the generic structure of an FI environment, which allows to outline the steps involved in an FI experiment. As outlined in Figure 3.2, many FI tools augment a preexisting *target system* – the execution environment running the actual workload – with an FI system.

In order to run the FI experiment for one fault from the fault-set F, the following steps – orchestrated by the *controller* component (Figure 3.2) – are necessary:

2 Benso and Prinetto [BP03] use the *fault model* (see Section 3.1.3) as a third fault-space dimension. Under the assumption that a specific fault model – for example, uniformly distributed single-bit flips in main memory – is selected beforehand, a two-dimensional fault space suffices to characterize faults.

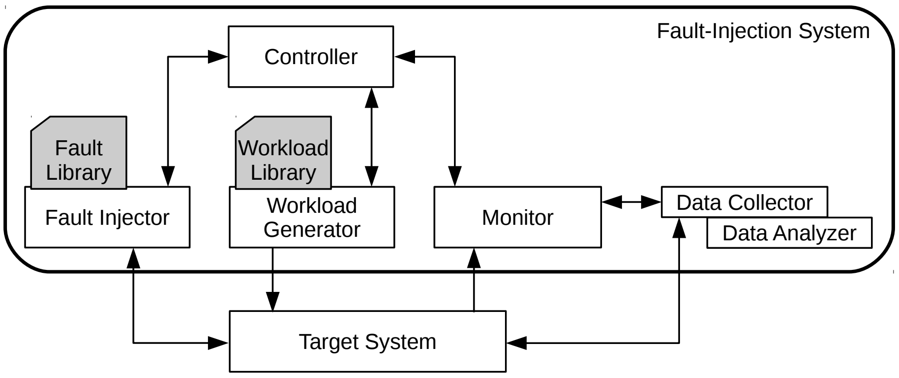


Figure 3.2: Basic components of an FI environment. *Adapted from [HTI97].*

- The *workload generator* supplies a workload – for example, a benchmark program with a predefined input – from its *workload library* to the target system. [HTI97]

- The *fault injector* starts the workload, runs the workload until the exact injection time is reached – an operation called *fast-forwarding* [PTAB14], – and injects a fault from its *fault library* at the specified fault location.

- The *monitor* component observes the workload execution on the target system, and instructs the *data collector* to record readouts (see Section 3.1.2.4) from the system, such as specific CPU states, serial output, or actuator activations. [HTI97]

- Usually after finishing an FI campaign, the *data analyzer* refines the readouts and distills measures, such as the fault-coverage factor metric (see Section 3.1.2.5). [HTI97]

In practice, the fault set F is often generated using a reference run of the target workload: the so-called *golden run*, in which no faults are injected.

Section 3.3.1.1 provides details on this so-called def/use pruning *process.*

3.1.2.3 *Activation Trajectories and Workloads*

The second component of the FARM model is the set of activation trajectories A that *"specify the domain used to functionally exercise the system"* [GRSRV06]. In other words, this means the input of the system in terms of communication messages, sensor inputs, or any other kind of input data chosen as a "representative" code-activation stimulus. Similarly, the workload itself – from a workload set W, which Goloubeva et al. [GRSRV06] propose to complement the FARM model – is supposed to be "representative" for the system's usage in the field.

Choosing a representative workload – often classic benchmark suites, such as MiBench [GRE$^+$01] or SPEC CPU, are used as workloads for FI campaigns – and input is a research topic of its own, and already well-known in the domain of functional software testing [MSB11]. In the context of FI, for example Sridharan et al. [SK09a], Benso and Di Carlo [BDC11], and Di Leo et al. [DLAS+12, DL13] analyze the problem of representative activation sets.

On a side note, the restriction to a specific set of workloads and inputs is a general limitation of FI. In this context, FI can be seen as a special case of program testing, which Dijkstra denoted *"a very effective way to show the presence of bugs, but [...] hopelessly inadequate for showing their absence"* [Dij72]. Consequently, FI results always – especially when they indicate "perfect" fault tolerance of the analyzed subject – must be taken with a grain of salt, as the validity of the conclusions drawn heavily depends on the real-world representativeness of the chosen workloads and inputs.

Throughout this dissertation, I will not address the intricacies surrounding workload and input selection. I assume a given, expertly chosen and representative workload and input.

3.1.2.4 *Readouts and Measures*

As the third FARM component, the set R of readouts from the system is defined as the raw information that is obtained from an observation of the system during each FI experiment in the campaign. This includes especially the system's output via communication channels, but potentially also relevant parts of the machine state, such as the recording of CPU exceptions. During the fault-less golden run (see above), the readout gets conserved as a reference to compare against.

3.1.2.5 *The Fault-Coverage Factor*

At the end of the FI campaign, all readouts can be compared against the golden-run reference. This leads to determining the set M of measures by classifying all experiment outcomes in a failure model, such as described in Section 2.3.2.4. Counting the number F of "failed" experiments – for example, with an SDC outcome – out of a total of N experiments, this also allows to statistically estimate the fault-coverage factor [BCS69] as already introduced in Section 1.2.6:

$$c_{\text{est}} = 1 - \frac{F}{N}.$$

The fault-coverage c estimates the probability that the system recovers after a fault occurred [BP03] – or, inversely, one minus the probability that a failure is observed under the condition that a fault occurred:

$$c_{\text{est}} \approx P(\text{Recovery}|\text{Fault}) = 1 - P(\text{Failure}|\text{Fault})$$

Unless N corresponds to the complete fault space, or in other words, the fault set was sampled, the fault-coverage factor estimation entails a statistical error that can be quantified as well. Assuming ...

- a total fault-space size $w = \Delta t \cdot \Delta m$, with Δt corresponding to the total number of possible injection times, and Δm meaning the total count of all possible injection locations,

CONFIDENCE LEVEL	CUT-OFF POINT ζ
80 %	1.28
90 %	1.64
95 %	1.96
98 %	2.33
99 %	2.58
99.5 %	2.81
99.8 %	3.09
99.9 %	3.29
99.99 %	3.89

Table 3.1: Confidence levels and corresponding cut-off points. *Adapted from [Tri02].*

- an estimate c_{true} for the "true" coverage,
- and a cut-off point ζ corresponding to a specified confidence level (see Table 3.1),

the margin of error calculates as [LCMV09]:

$$e = \zeta \sqrt{\frac{c_{\text{true}}(1 - c_{\text{true}})}{N} \frac{w - N}{w - 1}}.$$

The confidence level and margin of error is interpreted as such that if the sampling process is repeated with the same sample count, the true fault-coverage c_{true} lies within the *confidence interval* $c_{\text{est}} \pm e$ with a probability corresponding to the confidence level – for example, 95 percent for $\zeta = 1.96$, (cf. Table 3.1). [LCMV09]

Similarly, for a given error margin e, the number of necessary samples N can be calculated [LCMV09]:

$$N = \frac{w}{1 + e^2 \frac{w-1}{\zeta^2 c_{\text{true}}(1-c_{\text{true}})}}.$$

If c_{true} is unknown a priori – because the fault-injection campaign is intended to yield the first estimate c_{est} for it, – N can be conservatively maximized for the worst-case $c_{\text{true}} = 0.5$ [LCMV09].

For more details on sampling, coverage estimation, and statistical error, I refer the reader to the literature [PMAC95, Tri02, RKK⁺08, LCMV09].

3.1.2.6 *Other Measures*

Another metric very closely related to the fault-coverage factor is the *Architectural Vulnerability Factor* (AVF) introduced by Mukherjee et al. [MWE⁺03b, MWE+03a]. The AVF measures the “vulnerability” of hardware structures in a microprocessor, and is defined *“as the probability that a fault in that particular structure will result in an error”* [MWE+03b]. Assuming FI on the microarchitecture level (see Section 3.1.3), this corresponds to the complementary probability measured by the fault-coverage factor:

$$\text{AVF} = 1 - c = \frac{F}{N}$$

Similarly, Sridharan et al. [SK08b, SK09b] define the *Program Vulnerability Factor* (PVF) for *“any architecturally-visible resource”* [SK09b], such as CPU registers or memory. Consequently, ISA-level FI results can be used to calculate the PVF analogously to the AVF.

Besides the fault-coverage factor – and the related AVF and PVF metrics, – FI can also help measure other dependability metrics. One widespread metric is *latency*, particularly *fault latency*, the propagation delay between injection of the fault, and its activation as an error [BP03] (see Section 2.2.2), and *detection latency*, the delay between injection of the fault, and its detection by an EDM/ERM.

Other dependability metrics, such as the MTTF, availability, or reliability (see Section 2.2.4), are not widely used in the context of FI experiments, because the radically increased fault probability impedes their direct measurement.

3.1.3 *Fault Models*

As described in Section 2.2.2, a hardware fault – for example, indirectly caused by cosmic rays (see Section 2.3.2) – can turn into an error on one layer of abstraction, escalate to a failure on that layer, and propagate to the next layer. A failure on layer n is a fault from the perspective of layer $n + 1$.

In the context of an x86 microprocessor, a concrete example for such a propagation chain would be the following:

- On the *transistor layer*, a transistor faultily switches – for example, due to ionization effects caused by a neutron impact.

- The failing transistor is part of a logic OR gate on the *gate layer*. Both gate inputs are *zero* at this moment, so the transistor failure is not masked: it escalates as a gate failure to the gate’s output as a logical *one*, instead of the correct zero value.

- The failing OR gate is part of the ALU on the *microarchitecture layer*. The ALU is in the progress of adding two integer numbers, and the failing gate flips a single bit in the output of the calculation.

- On the *Instruction-Set Architecture* (ISA) *layer*, an `ADD %eax, %ebx` instruction is in progress. The corrupted calculation result from the microarchitecture ends up in the CPU register `EAX`, effectively flipping a single bit in this register – compared to the correct result.

- Seen from the *high-level language layer* perspective of a C program controlling the *Anti-lock Braking System* (ABS) of a car, the integer variable accumulating the wheel-rotation angle over a period of time is temporarily located in the `EAX` register. The failing ISA register `EAX` – and, hence, the bit-flip in the angle-accumulating variable – leads the program to the wrong conclusion that a front wheel of the car is blocking, and releases the brake.

- In consequence, a fault on the transistor layer recursively escalates to a failure on the ABS *system layer* as a whole.

Unfortunately, modeling a computing system at a level of detail that allows to trace faults from the transistor up to the system layer is normally unfeasible for the purpose of SIHFT test, measurement, or analysis.[3] One reason is the complexity involved, resulting in immense computation efforts necessary for low-level simulations, entailing extremely long simulation times [CMC$^+$13]; another is the prevalent unavailability of low-level models for most microprocessors and memory technologies on the market.

Consequently, FI normally uses higher-level system models, and injects high-level faults that emulate lower-level propagation, for example on the ISA layer. This abstraction process from low- to high-level faults forms a *fault model*, and influences the FARM model's fault space: The fault distribution regarding both injection time and location changes depending on the modeled low-level faults. If, following the above example, faults are assumed to originate (only) from transistors in the ALU, a simple ISA-level fault model would restrict the fault space to single-bit flip injections in the output registers (location dimension) of instructions that exercise the ALU (time dimension).

In the case of SIHFT evaluation, FI on the ISA layer is relatively common. The primary reasons are:

1. Hardware prototypes are accessible – and, hence, injectable – on the ISA level, but usually not below when using debugging hardware (see Section 3.2.2.3).

2. Software running on the device itself can also only access state reachable from the ISA level. Hence, FI implemented in software that runs on the same device as the target workload – known as *Software-Implemented Fault Injection* (SWIFI) (see Section 3.2.3) – is per definition also limited to ISA-level FI.

3. ISA-level device models – though, not necessarily timing-accurate – exist for most if not all microprocessors, and their simulation is relatively fast [CMC$^+$13] (see Section 3.2.4).

Additionally, SIHFT – just as SWIFI – can only detect and correct errors on the ISA level. Consequently, the injection of faults on lower levels is dispensable – if sufficiently accurate ISA-level fault models exist.

3 Hardware-based FI is an exception to this, as the *real* system is used and does not need to be modeled (see Section 3.2.2).

3.1.3.1 *ISA-Level Memory Fault Models*

As effects of cosmic radiation are uniformly distributed in space and time, regular low-level device structures in memory chips such as DRAM and SRAM fortunately allow to create relatively simple high-level fault models. To emulate direct hits in memory structures resulting in transient faults, the literature relatively consistently[4] uses a fault space with uniform distribution of FI times and locations, and focuses on single-bit faults.

The validity of this fault model has repeatedly been confirmed by field studies in the last few years, for example Sridharan et al. [SSD$^+$13] report a uniform fault distribution in DRAM devices, a predominance of transient over permanent faults, and more single-bit than multi-bit faults. Dixit and Wood [DW11] also observe a dominance of single-bit faults, but – as many other studies – see multi-bit faults on the rise. Additionally, Henkel et al. [HBD$^+$13] come to the conclusion that SRAM memory errors are more likely than errors in CPU logic:

> *"In general, SRAMs are more vulnerable to soft errors than logic, since memory cells lack transient masking mechanisms [...]"* [HBD$^+$13]

Similarly, Dixit and Wood note that *"the total number of logic cell errors is relatively small compared to the total number of memory cell errors"* [DW11].

To summarize, memory faults are still relatively common [MWKM15] – and, according to some sources, even dominate CPU faults [DW11]. As relatively simple, realistic fault models exist to emulate them, I will focus on uniformly distributed, single-bit transient faults in memory in this thesis. Additionally, in some cases I will approximate multi-bit faults by eight-bit burst faults, in which all eight bits at one memory address are flipped at once; for example, the observation of Li et al. that *"word-wise multi-bit failures are quite common"* [LHSC10] can serve as a justification for this fault model. The RAMpage case study in Chapter 4 also investigates *permanent* memory faults, modeled as single "stuck-at" bits in memory [NTA78] (see Section 4.4).

3.1.3.2 *Independent Faults and the Single-Fault Assumption*

In both Chapter 1 and the previous sections within this chapter, I silently assumed that in one FI experiment, only one fault is injected. But, in theory, with the aforementioned fault model of uniformly distributed, independent and transient single-bit flips in main memory, *a single run* of a simple run-to-completion workload can theoretically be hit by *any* number of independent faults. Multiple faults can occur at arbitrary points in time, and affect different bits in memory. The following considerations serve as justification that the single-fault assumption is nevertheless valid.

4 With the exception of tools that implement the def/use pruning optimization, and do not apply equivalence-class weights afterwards; see Section 3.3 and Chapter 7 for details on this issue.

In Figure 3.1 on page 50, each black dot denotes a possible time[5] (CPU cycle) and location (memory bit) coordinate where a fault can hit the workload run and flip a bit in memory, affecting the stored value throughout the subsequent cycles until it gets eventually overwritten. Assuming an arbitrary number of faults per run creates a fault space with one FI experiment *for every subset* of these hit coordinates – with a cardinality growing *exponentially* with both time and location axes.

In reality, though, the probability for a single-bit flip occurring at one bit in main memory within the time span of one CPU cycle – e.g., the probability for flipping bit #3 exactly in the time frame between cycle 4 and 5 in Figure 3.1 – is extremely low. Using the mean $g = 0.057\,\mathrm{FIT/Mbit}$ out of three per-Mbit SERs from recent large-scale DRAM studies (0.061 FIT/Mbit [LHSC10], 0.066 FIT/Mbit [SL12], and 0.044 FIT/Mbit [SSD⁺13]), for a single bit, the soft-error rate per nanosecond – assuming a CPU clock rate of 1 GHz, or one cycle per ns – can be calculated as

$$\begin{aligned} g &= 0.057\frac{\mathrm{FIT}}{\mathrm{Mbit}} = \frac{0.057}{10^9\,\mathrm{h}\cdot 10^6\,\mathrm{bit}} \\ &= \frac{0.057}{10^9\cdot 3600\cdot 10^9\,\mathrm{ns}\cdot 10^6\,\mathrm{bit}} \approx 1.6\cdot 10^{-29}\frac{1}{\mathrm{ns}\cdot\mathrm{bit}} \end{aligned}$$

The probability of *one* benchmark run being hit by $k = 0$, 1, 2, or more independent faults can be calculated using the binomial distribution. Nevertheless, for such an extremely low fault probability, it can be well approximated using the Poisson distribution (assuming the occurrence of faults is a Poisson process [MW79, Nor96, LSHC07]) [Tri02]:

$$P_\lambda(k) = \frac{\lambda^k}{k!}e^{-\lambda} \tag{3.1}$$

The Poisson parameter $\lambda = gw$ is calculated using the aforementioned soft-error rate g, and the fault-space size $w = \Delta t\cdot\Delta m$ characterizing the benchmark by its runtime in CPU cycles Δt, and the amount of main memory in bits Δm it uses. Using concrete values for the benchmark runtime and memory usage, e.g., $\Delta t = 1\,\mathrm{s}$ – corresponding to 10^9 cycles for the assumed 1 GHz CPU – and $\Delta m = 1\,MiB = 2^{23}\mathrm{bit}$, yields the probabilities for k faults hitting one benchmark run listed in Table 3.2.

Unsurprisingly, for the magnitude of the parameter values, the probability that a workload run is *not hit at all* is extremely high. This zero-fault case naturally does not require any FI experiments. But, even more noteworthy, the probability for two or more hits is so much lower than for one fault hitting the benchmark's used memory Δm that these cases can be considered negligible.[6] Hence, at current – and tomorrow's – fault rates, and for short workload runtimes, it suffices to inject one fault per FI experiment.

5 Note that time is quantized in granules of CPU cycles, a common assumption for higher-level fault models.

6 Even at a hypothetical fault rate of $g = 10^{-20}$, nine orders of magnitude higher than in the example, the distance between $P_\lambda(1\,\mathrm{Fault})$ and $P_\lambda(2\,\mathrm{Faults})$ is still more than 10^4.

k	$P_\lambda(k\,\text{Faults})$	k	$P_\lambda(k\,\text{Faults})$
0	0.999 999 999 999 867	3	3.905×10^{-40}
1	1.328×10^{-13}	4	1.297×10^{-53}
2	8.821×10^{-27}	…	…

Table 3.2: Poisson probabilities for $k = 0, 1, 2,$ or more independent faults hitting one benchmark run.

3.1.3.3 *ISA-Level CPU Fault Models*

In contrast to memory fault models (see Section 3.1.3.1), accurate ISA-level fault models for transient hardware faults within CPU structures are hard to come by, and are still subject to active research [CS90, KKA93, YS96, RLX+98, KIR+99, KMJM08, LRK+09, MKT+09, MKT+11, Kor12, CMC+13]. Faults within the sequential logic of the CPU can – as in the example at the beginning of Section 3.1.3 – manifest as single-bit flips in CPU registers or memory, but can also cause much more complex effects, for example multi-bit flips [CMC+13] or spurious interrupts [Kor12], or bring the CPU to an unrecoverable deadlock [Kor12].

As a simple approximation, many FI tools emulate CPU faults by single-bit flips in CPU registers [HRS95, TI95b, BPRSR98b, CCDM+09, SBK10, DBG+11, LT13, TP13], although recent research [CMC+13] criticizes this fault model for its inaccuracy. Early approaches assume a uniform distribution in both time and location (register bits) [HRS95, PRSR98]. More recent tools restrict the injection times to points in time when a register is about to be read by an instruction ("inject-on-read") [BVFK05b, SBK10, DBG+11, SAJK15], or has just been written ("inject-on-write") [SSSF10b, DBG+11, LT13, TP13, SAJK15]. Some approaches also consider FI in memory – as well "on read" or "on write" – as a valid approximation for CPU faults [SAJK15].

As mentioned in Section 3.1.3.1, I focus on memory faults in this dissertation. Nevertheless, the FAIL* tool in Chapter 4 is also capable of injecting CPU faults, emulated by bit flips in CPU registers, as demonstrated in some of the publications referenced in Table 4.12 on page 145.

3.1.3.4 *Other Fault Models*

Besides hardware faults in memories and CPUs, the literature also describes fault models for other types of hardware, for example hard-disk controllers, or network interface adapters [PSDC07]. Some models specifically and more abstractly deal with communication faults in distributed systems [HRS95, DJM96, DJMT96, LMX04, LX07, GDJ+11]. Many approaches specifically deal with *software* faults (bugs), and how to emulate and inject them [MCV00, DM06, JSM07, Joh08, NCDM13, GKT13, Giu14, WPS+15, HRP15, PWSF15].

These fault models – especially software faults – are outside the scope of this dissertation.

3.2 FAULT-INJECTION TOOLS AND TECHNIQUES

FI tools can be categorized by the FI technique they implement, by the purpose they serve, or by the fault models they support. In the following, I provide a FI-tool characterization scheme, and then discuss different FI techniques with a focus on hardware faults. Each technique – on the top level divided into *hardware-based*, *software-implemented*, *simulation-based*, and *symbolic* FI – is accompanied by a survey of FI tools implementing it.

3.2.1 *Characterization*

FI tools and techniques are characterized by the degree they exhibit the following attributes [SBK10]:

REPEATABILITY is an FI technique's ability to inject a specific fault and obtain the same result when repeated.

CONTROLLABILITY designates the capability to control exactly when and where to inject a fault, and is a prerequisite to repeatability.

INTRUSIVENESS means the measurement alteration caused by the FI technique on the target system, sometimes also termed the *"probe effect"* [CRS99, RAA02]. FI is intrusive when, for example, the target system's code, data-structure layout and positioning in memory, or timing must be changed for injecting faults.

OBSERVABILITY denotes the degree of making the effects of an injection observable and measurable.

REACHABILITY addresses an FI technique's capability to reach fault locations within the system. In other words, it denotes how much of the CPU and periphery state is accessible for injections.

For comparison, I add the following properties:

SCALABILITY describes the technique's ability to easily scale up to large quantities of FI experiments, if the workload complexity or analysis type demands it.

REALISM means the closeness to reality of both the target-device model, and possible fault models.

COST includes the expenses for prototype hardware, and the FI setup.

In the following sections – describing different FI techniques and tools from the literature – I will mention these attributes where applicable.

3.2.2 *Hardware-Based Fault Injection*

Hardware-based FI techniques rely on dedicated injection hardware – hence their name, – but also *inject into* real hardware – for example, development prototype boards. As both CPU(s) and memory devices are available as real hardware, no device model – in the sense of Section 3.1.3 – is required.

Surveying the literature, hardware-based FI techniques can be divided into techniques *without* contact between the injection hardware and the target device, and those *with* contact. FI without contact uses different types of radiation to affect the target, while contact-based approaches are directed towards a microprocessor's power-supply or data pins, or inject faults through an existing debugger interface, also known as a *Test-Access Port* (TAP).

3.2.2.1 *Radiation-based Fault Injection*

Due to closeness to the root causes of soft errors in reality, early hardware-based FI solutions emulating hardware faults were based on experiments with radiation sources, and are partially still state of the art. The general idea is to expose an existing target-system prototype – possibly including hardware- or software-implemented fault-tolerance implementations – to a radiation source that resembles or emulates natural causes for soft errors, but at a fault rate orders of magnitude higher than in reality.

For example, Gunneflo et al. [GKT89], Karlsson et al. [KGLT91, KLD$^+$94, KFA$^+$94, KFA$^+$95a, KFA$^+$95b], Miremadi and Torin [MT95], or Arlat et al. [ACK$^+$03] expose prototype hardware to heavy-ion radiation – in most cases originating from a decaying Californium-252 source. The goal of these studies was either to test SIHFT techniques, for example the fault-tolerance mechanisms implemented in the MARS operating system [KDK$^+$89], or to compare radiation-based FI with other FI techniques. Similarly, Karlsson et al. [KFA$^+$94, KFA$^+$95b, KFA$^+$95a] and Arlat et al. [ACK$^+$03] use *Electromagnetic Interference* (EMI) to artificially cause soft errors. Newer studies rely on neutron-beam facilities [BGGO03], such as available at *Los Alamos Neutron Science CEnter* (LANSCE), that are capable of emulating the energy spectrum of atmospheric neutrons, but at a rate increased by a factor of about 10^8 [CMR15]. Using neutron beams, Granlund et al. [BGGO03, GGO03b, GGO03a, GGO04, GO05, GO06] and Dixit et al. [DW11] study SERs of different generations of SRAM devices. Rech and Carro [RC13], Oliveira et al. [ORQ$^+$14], and Santini et al. [SRN$^+$14, SRCRW15] measure SIHFT effectiveness under neutron influence.

Although triggering realistic fault scenarios, the main disadvantage of these radiation-based approaches is that FI experiments have a very low controllability: Neither the time *when* faults are injected, nor the location *where* state alterations occur, can be defined before – or even only determined after – the injection by the experimenting user. As a consequence, they are not deterministically repeatable [ZAV04]. These properties – controllability and repeatability – are especially important for reproducing bugs in the implementation of SIHFT, for collecting fine-grained analysis results, and for systematic so-called "pruning" techniques to reduce the amount of experiments to conduct (see Section 3.3). Additionally, experiments with radiation sources are extremely expensive regarding both money[7] and time, and handling of radioactive materials like Californium-252 is relatively delicate [KLD$^+$94]. Nevertheless, radiation-based techniques share a negligible intrusiveness and good reachability, as radiation can trigger faults in any part of the target device.

Another radiation-based technique is laser FI, in which short pulses of focused laser light are sent onto exposed chip surfaces. Buchner et al. [BWK$^+$87] use a pulsed picosecond laser to cause SEUs in a logic circuit and a CMOS SRAM device. Samson et al. [SMF97, SMF98] and Moreno et al. [MFS98] use lasers to test hardware-implemented fault-tolerance schemes. Walters et al. [WZF13] test application-level control-flow assertions (see Section 2.4.3.5) and watchdog timers on the NASA SpaceCube platform.

Laser FI shares the low intrusiveness and good reachability of other radiation-based techniques, but offers strongly improved controllability and repeatability: *"[L]aser-induced faults can be finely controlled in space and time; single laser pulses can be injected into a system in order to test single event effects, and a laser can be readily focused on a circuit of interest with a spot size on the order of a micron"* [WZF13]. Nevertheless, the setup costs for laser FI are high, and require intimate knowledge on the exact on-chip location of the hardware structures to inject in. Additionally, especially the precision in the space dimension – *"with a spot size on the order of a micron"* [WZF13] – is clearly insufficient for contemporary circuits produced with structure sizes of a magnitude of nanometers.

Unlike described in Section 3.1.2 and Section 3.1.3, radiation-based FI techniques – especially neutron or heavy-ion radiation – do not explicitly define a fault list or a fault model: These techniques provoke faults in the semiconductor hardware that are very close – or identical – to those occurring in reality (see Section 2.3). Due to this realism, radiation-based FI – including laser FI – is still considered *"the gold standard for assessing radiation-hardness assurance"* [QBRB13]. On the downside, these techniques cannot determine whether no faults, or one or more faults were injected during a workload run – although readouts are collected as usual. This eliminates the possibility to calculate the fault-coverage factor $c_{\text{est}} = 1 - P(\text{Failure}|\text{Fault}) = 1 - \frac{F}{N}$ (see Section 3.1.2.5), which is calculated under the condition that one fault occurred.

Instead, the reliability $R_{\text{rad}}(\Delta t)$ *under increased radiation conditions* can be directly calculated, using the number of workload runs N, and the number of failed runs $F(\Delta t)$ (see Section 2.2.4):

$$R_{\text{rad}}(\Delta t) = 1 - P(\text{Failure}) = 1 - \frac{F(\Delta t)}{N}.$$

7 For example, neutron-beam experiments at LANSCE cost about 10,000 USD per day [Rec15], and require special training for the person responsible for experiment setup and execution.

If the exact radiation intensity is known, this value can be corrected to yield $R(t)$ under real-world conditions. This can be achieved by reducing $P(\text{Failure})$ by a linear factor [Muk08, NIS].

3.2.2.2 *Power-Supply Disturbance and Pin-Level Fault Injection*

In contrast to radiation-based methods, contact-based FI techniques inject faults by attaching an injection device directly to the system under test. The injection itself is executed by disturbing the power supply, or by altering the data transferred at the input and output pins of the microprocessor.

Karlsson et al. [KGLT91] introduce short voltage drops at the power-supply pin of the target microprocessor, and primarily observe control-flow errors in the system. Similarly, Miremadi et al. [MT95] test a control-flow SIHFT technique [MKGT92]. Rajabzadeh et al. [RMM04] and Tummeltshammer et al. [TS09] test dual-core lockstepping solutions under the influence of power-supply disturbances.

While power-supply disturbance only affect the power-supply pin of a microprocessor, generic pin-level FI techniques target all (digital) processor pins. There exist two approaches to pin-level FI: *forcing* – the application of faults using probes, – and *insertion*, in which the processor is inserted in a special circuit isolating the chip from the system. For example, MESSALINE by Arlat et al. [ACL89, AAA+90, ACC+93] supports both forcing and insertion, and can inject stuck-at faults, inversion, bridging, and open-contact faults. RIFLE by Madeira et al. [MRMS94] supports a similar set of injectable faults (using the insertion variant), and claims to support deterministic – and, hence, repeatable – FI experiments. Martínez et al. describe AFIT [MGM+99], which uses forcing for transient, intermittent, and permanent stuck-at faults at the pin level.

Power-supply disturbance, and especially pin-level FI, have much better controllability and repeatability than the mentioned radiation-based techniques – for example, RIFLE explicitly claims these properties [MRMS94], – but at the cost of reachability: Faults cannot be injected into the internal state of the system, but only at the interface between the microcontroller and the remaining system, which strongly restricts possible fault models. Not only for these reasons, Santos and Rela describe pin-level FI as being *"obsolete due to the increasing complexity and greater miniaturization of digital circuits"* [SZR03].

3.2.2.3 *Test-Access Port Fault Injection*

Another contact-based FI technique uses the *Test-Access Port* (TAP) – a microprocessor interface usually found in embedded systems – to inject faults. Although a TAP is intended for debugging, firmware uploads, and device-hardware testing, its capability to control the CPU's execution and to access register and memory contents also facilitates ISA-level FI [SZR03].[8] Widespread

8 TAPs also allow the injection of pin-level faults [SZR03]. I will not discuss this capability further, as this dissertation focuses on faults in memory (see Section 3.1.3).

TAP standards are JTAG [MT90], Nexus [II99], and the Freescale-specific BDM [How96].

A classic example for FI via the JTAG TAP is FIMBUL [FSK98], which was later renamed to GOOFI [AVFK01, VAS$^+$05]. Triggered by breakpoints on static instructions or memory accesses to specific addresses, FIMBUL and GOOFI inject single-bit flips into the memory and scan registers of the Thor CPU. Similarly, Benso et al. [PRSR98, BRSR99, RSR99] inject bit-flips in main memory and CPU registers via BDM in their FlexFi tool. Later versions of Xception [MHCM02, MHB$^+$05] are also reported to support an unnamed TAP standard for FI.

Newer FI tools, such as Yuste et al.'s INERTE [YdAL$^+$03, YRLG03], Fidalgo et al. [FGAF06], and Skarin et al.'s GOOFI-2 [SBK10, Ska10], support the Nexus debugging interface for injection. Yuste et al. and Fidalgo et al. inject bit-flips in the memory without halting the target system – a Nexus specialty that allows the workload to meet real-time conditions, presumably at the price of FI-timing accuracy. Skarin et al. (GOOFI-2) instead halt the CPU before injecting into registers or memory [SBK10], similar to BDM- or JTAG-based FI tools.

FI via a TAP offers very good controllability and repeatability, and a low intrusiveness – especially regarding workload timing in program phases without FI. It combines these advantages with a reachability sufficing for the ISA-level fault models described in Section 3.1.3. Disadvantages are the additional cost for debugger interfaces and target-device prototype hardware, and consequently a low scalability. Additionally, a notable slowdown of the target device is incurred with TAP-based FI when compared to fault-free runs – for example, Rebaudengo et al. [RSR99] report runtimes up to 97 times slower. Rebaudengo et al. [RSR99] identify the setup phase including workload download and initialization, and especially the *fast-forwarding* operation (see Section 3.1.2.2 and Section 3.3.2.1), as the primary bottlenecks.

3.2.3 *Software-Implemented Fault Injection*

To counter the high cost, and low repeatability, controllability and scalability of hardware-based FI techniques, *Software-Implemented Fault Injection* (SWIFI) was introduced in the 1990s and is still quite popular until today. SWIFI techniques can be distinguished in *pre-runtime* and *runtime* SWIFI.

3.2.3.1 *Pre-Runtime SWIFI*

Pre-runtime SWIFI techniques inject faults in the workload's data or code – usually by modifying the workload image – before it starts to run on the target system. For example, Kanawati et al.'s FERRARI tool [KKA95] injects single-bit flips into machine instructions (but is also capable of runtime SWIFI, see below), and was, for example, used to evaluate Alkhalifa et al.'s ECCA CFM approach [ANKA99] (see Section 2.4.3.5). Similarly, Fuchs et al. [Fuc96] use a nameless SWIFI tool to inject up to ten single-bit flips into code and data segments of the MARS [KDK$^+$89] operating system. GOOFI and GOOFI-2 [AVFK01, VAS$^+$05, SBK10, Ska10] are, among other FI techniques, also capable of pre-runtime SWIFI.

More recently, Schiffel et al.'s EIS [SSSF10b, Sch11] and LLFI by Thomas, Wei, Lu et al. [TP13, WTLP14, LFW$^+$15] apply pre-runtime SWIFI on the LLVM [LA04] IR level. The primary advantage is independence from a particular target machine, and Wei et al. [WTLP14] claim a good match with machine-code level pre-runtime SWIFI – at least regarding SDC-causing faults.

With the aforementioned advantages regarding cost, repeatability, controllability, and scalability, pre-runtime SWIFI also offers a low intrusiveness: As the fault is injected before starting the target device, the workload execution does not need to be interrupted at runtime. An obvious disadvantage is a low reachability – only machine instruction op-codes and static data can be injected into, – and consequently, a debatable fault-model realism.

3.2.3.2 *Runtime SWIFI*

In contrast to pre-runtime SWIFI, *runtime* SWIFI is based on additional software running on the target device alongside the actual workload. This software injects a fault at runtime when a specific trigger condition – for example a specific point in time – is reached.

One of the first tools regularly cited to be implementing runtime SWIFI [VAS$^+$05, Ska10] is FIAT by Barton et al. [BCSS90] and Segall et al. [SVS$^+$95]. The tool injects single- and multi-bit faults into code and data of the target workload, but the original paper stays rather vague on how the injection is actually triggered at runtime.[9] Around the same time frame, Kao et al.'s FINE [KIT93] and its distributed-system variant DEFINE [KI94] implement a system call to inject bit flips into code and memory of the UNIX kernel at runtime, with the goal of studying the fault's propagation throughout the kernel. Similarly, Hiller et al.'s PROPANE [HJS02b, Hil02] aims at studying the error propagation in embedded software. By means of source-code transformations, PROPANE injects several types of variable errors: bit-flips, but also variable-specific fixed values, and offsets. Hiller et al. subsequently distill several different module-level metrics from the results – among others, the module's *exposure* to errors, its *permeability*, its *impact* on the remaining modules, and its *criticality*, – and use them for EDM placement in an aircraft arresting system[10] [HJS01, HJS02a, HJS04]. Dasilva et al.'s commercial Exhaustif tool [DML$^+$07, DGCM$^+$09] also targets (heterogeneous) embedded systems, and injects into CPU registers and memory.

Han et al.'s DOCTOR [HRS95] injects transient, intermittent and permanent single- and multi-bit flips in CPU registers and memory, but also specific communication-related faults, for example lost or duplicated messages.

9 In fact, Section III in the original paper [BCSS90] can be read in a way that FIAT really uses pre-runtime SWIFI to emulate runtime effects.

10 *"The target system is a medium sized embedded control system used for arresting aircraft on short runways and aircraft carriers. The system aids incoming aircraft to reduce their velocity, eventually bringing them to a complete stop."* [HJS01]

Tsai et al.'s FTAPE [TI95b, TI95a] uses different fault distributions over time, and injects single- and multi-bit faults in CPU registers, memory, and I/O operations to compare different fault-tolerant systems. Stott et al.'s distributed NFTAPE variant [SFKI00] was subsequently used by Gu et al. [GKI04] to analyze the error sensitivity of Linux. Rodríguez, Arlat et al.'s MAFALDA [RSFA99, AFRS02, RAA02] was specifically designed to assess the robustness of a microkernel OS, and injects bit flips in code and data memory, but also in a selective way in the system-call interface. MAFALDA was, for example, used to measure and compare executable assertions in the Chorus microkernel [SRFA99] (see Section 2.4.3.3).

Some runtime SWIFI approaches *only* aim at specific software interfaces, further abstracting from actual hardware – or software – faults behind these interfaces. Koopman et al. [KSD+97] target the system-call interface and thereby compare the robustness of several commercial operating systems. Broadwell et al.'s FIG [BST02] and Marinescu et al.'s LFI [MC09, MBC10] inject into the application/library interface, and Winter et al.'s simFI [WTSS13] and GRINDER [WPS+15, Win15] address different OS-kernel internal interfaces.

Only a few all-rounder FI tools stand out as they must be mentioned in multiple technique categories. For example, FERRARI [KKA95], already mentioned for its pre-runtime SWIFI capabilities, also injects bit flips at runtime in the workload's code triggered by software traps – either reaching a specific point in time, or a static instruction. Similarly, Xception by Madeira, Carreira, Maia et al. [MCS95, CMS95, CMS98, MHCM02] – already mentioned in Section 3.2.2.3 for its TAP-based FI capabilities – emulates transient and permanent hardware faults by bit flips, stuck-at-zero, and stuck-at-one faults in CPU registers or memory. Xception's FI routines on the target system are triggered by hardware exceptions available on the supported PowerPC MPC601 CPU: accesses to specific memory addresses, the execution of specific static instructions, or timeouts measured in dynamic instructions or wall-clock time. Another all-rounder, EXFI [BPRSR98b, BPRSR98a] and its successor FlexFi [BRSR99] – the latter also listed among TAP-based FI tools – relies on trace-mode execution capabilities of modern CPUs[11] to inject transient single-bit flips in CPU registers and memory at specific dynamic instruction counts. The GOOFI tool series, already cited for its TAP-based FI and pre-runtime SWIFI capabilities, also supports runtime SWIFI since the GOOFI-2 version [SBK10, Ska10]. Among other details regarding FI optimizations (see Section 3.3), GOOFI-2 adds multi-bit flips in CPU registers and memory at runtime to GOOFI. Giuffrida et al.'s EDFI [GKT13, Giu14] combines pre-runtime and runtime SWIFI, but specifically targets software faults.

Compared to pre-runtime SWIFI, runtime SWIFI trades low intrusiveness for reachability and realism. The modified workload, including additional instructions running on the target system, are intrusive in both space and

11 In this case, Benso et al. used the Motorola M68040's capability to trap after the execution of every machine instruction [BPRSR98a].

time: The workload's behavior and – especially when the SWIFI approach uses the processor's tracing mode – its timing changes compared to the original workload. Nevertheless, runtime SWIFI is one of the most popular FI techniques until today, not least due to its low cost and its machine-state reachability.

3.2.4 *Simulation- and VM-Based Fault Injection*

All mentioned hardware-based FI techniques (see Section 3.2.2) as well as SWIFI (see Section 3.2.3) assume an existing prototype of the target device that can run the workload. In early development phases, a prototype may often not yet exist, or access to it is congested by developers dealing with other, functional properties of the target workload. In simulation-based and *Virtual Machine* (VM)-based FI, the prototype is replaced by virtual, software-simulated hardware, in which faults are injected. I will not address the differences between simulation-based and VM-based FI within this section, as the technical details how the workload's machine instructions are executed on the host machine are of little relevance for the points made here.[12]

Some simulation-based FI tools operate on machine-detail levels below ISA level. For example, Jenn et al.'s MEFISTO [JAR+94], extended by Boue et al.'s MEFISTO-L [BPC98] to facilitate testing of hardware-implemented fault-tolerance mechanisms, inject faults into an *Register-Transfer Level* (RTL)-level and a VHDL model of a 32-bit CPU. They insert *mutants* – structural modifications – into the model, but also inject transient, intermittent, and permanent faults into VHDL signals and variables by using specific simulator commands. Sieh et al. [STB97] also work on the VHDL level with their VERIFY tool, but extend VHDL itself to describe the hardware behavior under faults. To counter the extremely low speeds of low-level simulators, tools like Pellegrini et al.'s CrashTest [PCZ+08] or the nameless tool by Daveau et al. [DBG+09] emulate gate-level machine models on FPGAs, and parallelize FI experiments.

Nevertheless, the glacial performance of low-level simulators – and, not least, the prevalent unavailability of low-level device models (see Section 3.1.3) – leads to SIHFT usually being evaluated at higher machine-abstraction levels, commonly using a behavioral ISA-level model. For example, FAUmachine [PSDC07, SPS09] is an ISA-level x86 simulator built specifically for FI into CPU registers and memory, but also into other PC components, such as network-interface controllers or hard disks. FAUmachine also provides elaborate scripting capabilities to automate interactions with the target workload, for example to repeatedly test the installation of an operating system.

Unlike FAUmachine, most other simulation-based high-level FI tools – at least partially – rely on existing simulator software, and extend it by FI ca-

12 Some sources also distinguish between CPU *simulation* and *emulation*, ostensibly differentiating between detailed and less detailed, behavior-level simulations. Without a strict definition which is which, I will use the term *simulation*, even if, for example, Bellard calls his QEMU simulator a *"machine emulator"* [Bel05].

pabilities. Presumably owing to its popularity, many FI tools are based on the open-source QEMU [Bel05] simulator, for example David et al.'s Qinject [DCCC08], DeBardeleben et al.'s SEFI [DBG+11] and its successor F-SEFI [GDBF14], Chylek et al. with QEFI [CG12, Chy14], Xu and Xu's [XX12], Di Guglielmo et al.'s [DGFFP13], and de Aguiar Geissler et al.'s [dAGLKPS14] nameless tools, Wanner et al.'s VarEMU [WEL+13], or Höller et al. with FIES [HSK+14, HMR+15, Sch15a].

Besides QEMU, many other simulators are used for FI purposes. Xu et al.'s CriticalFault [XL12] and Hari et al.'s Relyzer [HANR11, HANR12, HANR13] inject faults into the commercial Simics simulator. Li et al.'s SmartInjector [LT13] enhances the SimpleScalar simulator with FI capabilities, Parasyris et al. extend Gem5 [BBB+11] in their GemFI tool [PTAB14], and Winter et al.'s simFI [WTSS13] injects into Xen [BDF+03] virtual machines. Terasa et al.'s and Heing et al.'s FITIn [TS13, HBKS14] extends Valgrind's CPU simulator by FI primitives.

The primary disadvantage of simulation-based – and likewise, VM-based – FI is its reduced workload-execution speed, compared to the previous FI techniques using real prototype hardware. Especially low-level simulators tend to be many orders of magnitude slower than the hardware they model. The device-model realism of simulation-based FI naturally depends on the detail level of the simulator, which also influences the FI technique's reachability: While FI in gate-level simulations can reach arbitrary gates and flipflops within any simulated component of the CPU, FI in a behavioral, high-level simulation often only reaches ISA-visible CPU registers and memory cells.

On the positive side, simulation-based FI has several advantages over other FI techniques. Particularly, it avoids the intrusiveness of runtime SWIFI, and offers high controllability and repeatability. Simulation-specific capabilities like checkpointing open up optimization potential impossible to achieve with other FI approaches: Checkpoints can, for example, be used to speed up experiments (see Section 3.3). Additionally, and despite the reduced execution speed, the scalability of simulation-based FI excels other FI techniques: Experiments can be run on *any* available hardware – for example, a computing cluster – instead of costly device prototypes.

3.2.5 *Symbolic Fault Injection*

Symbolic FI is a rather exotic category of FI tools. It descends from the model-checking domain, and combines the idea of symbolic execution with FI. Symbolic FI requires an accurate model of the software system, which can be hard to come by in pointer-afflicted programming languages like C or C++ that are normally used in the embedded-systems domain.

A recent example is Pattabiraman et al.'s SymPLFIED [PNKI08, PNKI13], a symbolic-FI framework for verifying error detectors in programs using symbolic execution and model-checking. Although an impressive feat, the approach is described to yield false positives, and its scalability is extremely limited: Nearly half of the analysis tasks did not run to completion due to the analysis complexity, even though Pattabiraman et al. only analyzed program input in the magnitude of merely hundreds of lines of C code. Hähnle and Larsson's KeY tool [HL06, LH07a] is, similarly, evaluated with only very small code examples. Alvaro et al.'s MOLLY [ARH15], reasoning backwards from correct system outcomes to look for flaws in the fault-tolerance implementation of a data-management system, is demonstrated on protocol implementations sized 5 to 41 lines of code.

Symbolic FI is quite tempting for the evaluation of SIHFT, as it can *"evaluate the consequences of* all *possible faults [...] for* all *possible system inputs"* [LH07a] and allows *"to formally prove properties of fault tolerance mechanisms"* [LH07a]. Nevertheless, its limitations regarding problem sizes and model inputs make the technique hard to use in practice. Additionally, symbolic FI only helps finding flaws in a fault-tolerance implementation, comparable to the "testing" purpose described in Section 3.1.1, but does not provide quantitative measurements for workload-variant comparison.

3.2.6 *Summary*

To summarize, the existing FI technique and tool landscape is quite diverse regarding the characterizing properties *repeatability*, *controllability*, *intrusiveness*, *observability*, *reachability*, *scalability*, machine and fault-model *realism*, and *cost* (see Section 3.2.1). Table 3.3 provides an overview which techniques excel at the different properties. Radiation-based FI is the only technique providing perfect fault- and machine-model realism, while TAP-based FI still uses real hardware and, hence, provides a realistic machine model. Among the techniques not requiring a hardware prototype, simulation-based FI provides the best mixture of positive characteristics, only having the disadvantage of slow simulation speeds.

3.3 FAULT-INJECTION OPTIMIZATIONS

As already outlined in Section 1.2.3, FI campaigns can be optimized by reducing the total campaign runtime $T_{\text{campaign}} = N \cdot T_{\text{experiment}}$, the product of the FI experiment count N and the (average) per-experiment runtime $T_{\text{experiment}}$. A campaign-runtime reduction can either be achieved by decreasing either N or $T_{\text{experiment}}$ [GRSRV06]. The following subsections discuss existing approaches for both methods.

3.3.1 *Experiment-Count Reduction*

Many studies leveraging FI for testing, measurement, and comparison purposes resort to *statistically sampling* fault locations [RKK+08, LCMV09]: A randomized selection of fault locations – usually in the thousands – is used to drive the FI campaign, until statistics predict a "good enough" probability for the overall result distribution to lie within the desired confidence interval

	HW-based			SWIFI			
Property	Radiation	Pin/Power-Supply	TAP	pre-runtime	runtime	Simulation-based	Symbolic
Repeatability	○	◐	●	●	●	●	●
Controllability	○	◐	●	●	●	●	●
Non-Intrusiveness	●	◐	●	●	○	●	○
Observability	◐	◐	◐	●	●	●	●
Reachability	●	○	◐	○	◐	◐/●	○
Scalability	○	○	○	●	●	●	○
Realism	●	○	◐	○	◐	◐	○
Low Cost	○	○	○	●	●	●	●

Table 3.3: Adding **scalability**, **realism** and **cost** to the categories described by Skarin et al. [SBK10] (see Section 3.2.1), the different FI techniques described in Section 3.2 exhibit different advantages and disadvantages when used for the injection of hardware faults. Legend: The *property* is ○ = not present, ◐ = moderately present, or ● = strongly present in the respective FI technique.

(Section 3.1.2.5 already discussed this). By sampling, the experiment count N can be reduced arbitrarily, trading experimentation efforts for lower error margins.

Nevertheless, for fine-grained analysis purposes (see Section 3.1.1), results on the complete fault space are desirable. As already discussed in Section 1.2.3, this is generally infeasible even for small applications, as – for the example of single-bit flips in memory – both space and time dimensions grow quickly, resulting in a large fault-space size w.

3.3.1.1 *Def/Use Pruning*

Recent FI techniques proceed with more sophistication than randomly sampling locations in the fault space, and are based on instruction and memory-access traces. These traces are created during the "golden run", which exercises the target workload without injecting faults, and, thus, serves as a reference for the expected program behavior (see Section 3.1.2.2).

Figure 3.3 exemplarily shows the memory-access information recorded during the golden run, extending the "raw" fault space from Figure 3.1 with these accesses. The dynamic instruction starting in CPU cycle 4, a *store*,

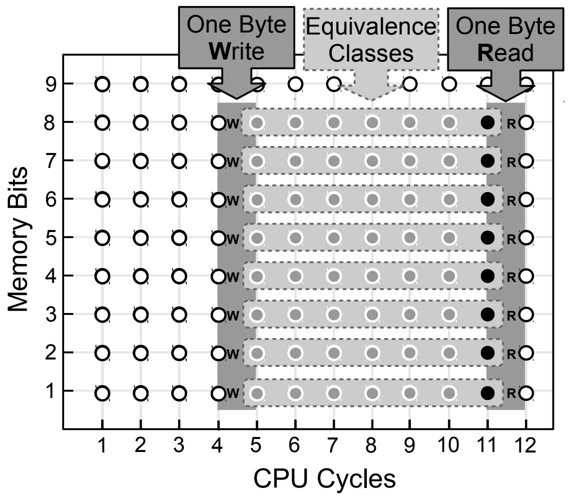


Figure 3.3: Single-bit flip fault space and def/use equivalence classes extracted from a program trace, reducing the FI experiments (dots) that need to be conducted. Fault injections at white coordinates (non-filled circles) can be omitted, as a fault there is overwritten or never read (dormant faults). A black dot represents a class of equivalent faults (light-gray coordinates) between the write and subsequent read instruction.

writes ("W") eight bits to main memory. The data is read ("R") back into the CPU in cycle 11, executing a *load* instruction.

Based on this kind of memory-access trace information, Smith and colleagues [SJPB95] and Güthoff and Sieh [GS95] are among the first concisely describing the classical *def/use analysis* for experiment reduction (termed "operational-profile-based fault injection" in the latter paper). It is so fundamental that it was subsequently reinvented several times, for example by Benso et al. [BRIM98, BP03], Berrojo et al. [BGC+02] ("Workload Dependent Fault Collapsing"), Barbosa et al. [BVFK05b, BVFK05a] ("inject-on-read"), and recently by Grinschgl et al. [GKS+12] and Hari et al. [HANR11, HANR12, HANR13].

The basic insight is that all fault locations between a *def* (a write or *store*, "W" in Figure 3.3) or *use* (a read or *load*, "R" in the figure) of data in memory[13], and a subsequent *use*, are equivalent: regardless of when exactly in this time frame a fault is injected there, the earliest point where it will be activated is when the corrupted data is read. Instead of conducting one experiment for every point within this time frame, it suffices to conduct a *single* experiment – for example at the latest possible time directly before the *load*, the black dots in Figure 3.3, – and assume the same outcome for all remaining coordinates within that *def/use equivalence class* (light-gray frames in Figure 3.3).

Similarly, all points in time between a *load* or *store* and a subsequent *store* without a *load* in between – the light-gray dots in the remaining fault-space coordinates left, right and above the marked equivalence classes in Figure 3.3

13 This technique also works for any other level in the memory hierarchy, such as CPU registers or cache memory.

– are known to result in “No Effect” without having to run experiments at all: Injected faults will be overwritten by the next *store* in all cases.

The result of the def/use pruning process is a partitioning of the fault space into equivalence classes. For some of these equivalence classes – those ending with a *load* (“R” in Figure 3.3), – a single experiment needs to be conducted each, while the remaining classes have an a priori known experiment outcome. From the $12 \cdot 9 = 108$ experiments in the illustrative example of Figure 3.3, only 8 remain after def/use pruning. A noteworthy side effect of def/use pruning is that it restricts FI to memory areas used at all by the workload, which alone can contribute significantly to experiment-count reduction.

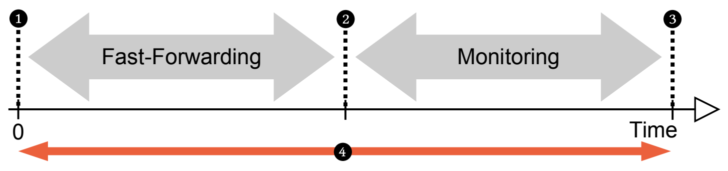
In real-world examples, def/use pruning – especially when applied to faults in memory – is extremely effective. As an example, Barbosa et al. [BVFK05b, BVFK05a] report two orders of magnitude experiment-count reduction (5.0×10^8 down to 7.7×10^6) for FI in registers for their jet-engine controller benchmark, and between four and five orders of magnitude (1.9×10^{11} down to 3.3×10^6) for faults in memory. Thus, for smaller workloads a full fault-space scan becomes feasible within a reasonable time frame, without any loss of precision regarding the result information on any point in the fault space.

3.3.1.2 *Fault-Equivalence Heuristics*

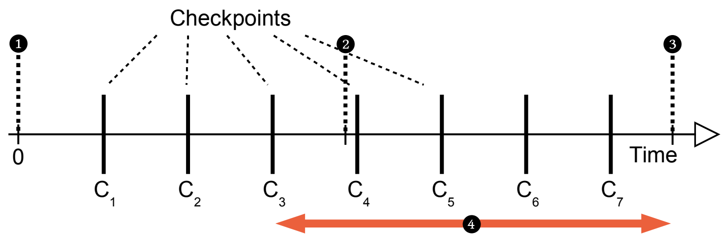
Although def/use pruning significantly reduces the number of experiments, the computational efforts for complete fault-space coverage are still far too heavy for most workloads. Only recently, more advanced fault-equivalence pruning techniques appear in the literature, leaving the realm of conservative, accuracy-neutral methods, in favor of heuristic approximations.

For example, Hari et al. [HANR11, HANR12, HANR13] describe the *Relyzer* tool, providing several heuristics that combine multiple def/use equivalence classes into larger groups. From each group, only *one* representing def/use equivalence class – the *pilot* – gets picked, and the experiment result is assumed to be identical for the remaining group members. Although rather effective, a major deficiency of this approach is the inflexibility regarding the result accuracy – the authors report an average accuracy of 96 percent for pilots representing their group – and experiment-count trade-off: If the result accuracy turns out too low, no alternative is offered; if the experiment count is still too high, the authors suggest sampling from the set of pilots. The latter, again, has the problem of not providing information on the complete fault space. Additionally, the grouping heuristics are based on complex control and data-flow analyses, SPARC platform specifics, and assumptions on the experiment result interpretation: Relyzer only differentiates between *benign* and SDC outcomes, while in many use cases more outcome types, or even quality thresholds on the output [DSE13, TP13], become relevant.

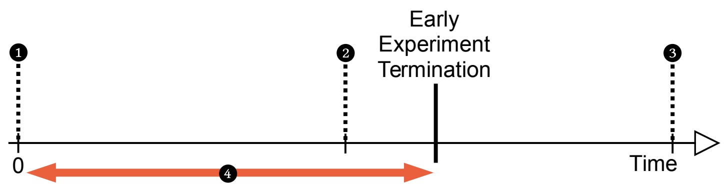
Li and Tan [LT13] describe similar pruning heuristics in their *SmartInjector* tool. They provide a slightly better experiment count reduction than *Relyzer*, but otherwise share the aforementioned drawbacks, including the



(a) FI-experiment phases under the single-fault assumption: *Fast-forwarding*, followed by *fault injection* ❷, and post-injection *monitoring*.



(b) FI-experiment speedup using checkpoints: By loading the closest checkpoint C_3 before the injection time ❷, the fast-forwarding execution time from the workload startup ❶ until C_3 can be saved.



(c) FI-experiment speedup by early experiment termination: By predicting the experiment outcome, the monitoring execution time until the regular workload termination can be saved.

Figure 3.4: Unoptimized FI-experiment phases, and speedup using checkpoints or early experiment termination. Legend: ❶ Workload startup, ❷ FI, ❸ regular workload termination, and ❹ actual code-execution time span on the target device.

single focus on SDCs, and a fixed trade-off between accuracy – reportedly 94 percent on average – and pruning effectiveness.

Chapter 5: Heuristical fault-space pruning technique.

To address these issues, including the fixed result accuracy, the restriction to only SDC outcomes, and the necessary complicated analysis steps, Chapter 5 describes my novel heuristical fault-space pruning technique. It reduces the computing efforts for complete fault-space exploration by *freely trading* the number of FI experiments for result accuracy, providing results for any number of *different outcome types*, and is based on the relatively simple idea of *fault similarity*.

3.3.2 *Experiment Speedup*

Hardware-implemented FI (Section 3.2.2) and SWIFI (Section 3.2.3) run the workload on real hardware.

Besides reducing the number of experiments, an FI campaign can also be accelerated by speeding up each single experiment. The easiest way to achieve experiment speedup is to simply choose an FI technique with a high workload-execution speed, such as the techniques running the workload on prototype hardware – or in a fast simulation environment, such as an FPGA-accelerated simulator [BGC⁺02, PCZ⁺08]. Depending on the availability of a hardware prototype or a fast simulator, and the requirements regarding fault-model realism and state reachability, this option may or may not be available.

In case FI techniques with high controllability and repeatability are used – regardless if for testing, measurement, or fine-grained analysis – more possibilities regarding experiment speedup arise. Recapitulating the FI-experiment steps from Section 3.1.2.2, an FI experiment can be divided into three phases (see Figure 3.4a):

- A *fast-forwarding* phase, advancing the workload until the exact injection time is reached,
- the *fault injection* itself, and
- a post-injection *monitoring* phase, collecting readouts while the workload finishes its run (or terminates abnormally).

The literature provides experiment-speedup approaches for both fast-forwarding and monitoring phases.

3.3.2.1 *Pre-Injection Speedup: Fast-Forwarding and Checkpointing*

With a high controllability, the *fast-forwarding* phase can demand a large time fraction of the FI experiment – even on real hardware with a high native workload-execution speed. In its simplest implementation, fast-forwarding can be implemented by single-stepping the target device. In the example of TAP-based FI via JTAG into contemporary ARM Cortex A9 development hardware, by far the most time is spent in the round-trip between the host PC and the development board: In this setup, this round-trip takes about 70 ms, and has to be taken for every single step – or, dynamic instruction – the workload is forwarded. For example, fast-forwarding the workload to the 400th dynamic instruction takes about 28 s with single-stepping [SBS14]. Fast-forwarding an ISA-level simulator is several orders of magnitude faster, but the basic problem stays the same.

Section 4.3.3 describes the ARM/JTAG setup that was used for this round-trip time measurement in detail.

A fast-forwarding speedup approach described by several authors is *checkpointing* [PRSRV00b, PRSRV00a, BGC⁺02, HVAN14, PTAB14]. Checkpoints represent loadable intermediate states at a priori chosen – and usually equidistant – dynamic instructions $N_1, N_2, \ldots$ (Figure 3.4b). They allow fast-forwarding to specific points in the execution trace, skipping all dynamic instructions up to N_i; from there, a fine-grained fast-forwarding technique like the aforementioned single-stepping forwards the final instructions to the target instruction.

Nevertheless, a checkpoint load costs a non-negligible amount of time, and consequently only pays off if this time does not exceed the time required for a simpler fast-forwarding technique. More importantly, *creating and storing* checkpoints consumes both time and storage space [BGC⁺02]. Therefore, the total number of checkpoints in use is limited, making this technique orthogonal to fine-grained fast-forwarding techniques such as the aforementioned single-stepping to navigate to the desired target instruction after the checkpoint load.

Chapter 6 discusses more sophisticated fast-forwarding alternatives to single-stepping, and presents my novel fast-forwarding technique that outperforms the state of the art by several orders of magnitude.

Chapter 6: A novel fast-forwarding technique.

3.3.2.2 *Post-Injection Speedup: Outcome Prediction*

Independently of speeding up workload execution until a fault is injected, accelerating the *monitoring* phase afterwards equally can reduce the total time for an FI experiment. The basic idea behind several approaches achieving this goal is to selectively terminate experiments before they finish regularly, saving the remaining execution time (Figure 3.4c). This can be done if the experiment result is predictable based on the machine state, or observations from previous experiments.

For example, Berrojo et al. [BGC+02] describe *dynamic fault collapsing*, speeding up FI experiments on the flip-flop level. After injecting a single-bit flip, their optimization approach repeatedly – with quadratically increasing time distances to the injection time – compares the machine state to the golden run. This comparison step discovers states that differ from the golden run by a single bit, rendering the original fault equivalent to a fault in the single, differing bit. If the result of the experiment with the equivalent fault has already been determined, the original experiment can be aborted and predicted to yield the same result; if the result is not known yet, the original experiment is finished, and its result is also valid for the equivalent fault, making an experiment unnecessary.

Similarly, Li and Tan's *SmartInjector* [LT13] predicts the outcomes of FI experiments on the ISA level, focusing on SDCs and benign outcomes. Instead of comparing the complete simulator state, SmartInjector's outcome-prediction heuristic compares the output or behavior of statically determined masking, output, or branching instructions with the golden run. Li and Tan report a simulation-time reduction of 19 percent throughout their benchmarks. Following the same basic idea, Hari et al.'s *GangES* [HVAN14] groups FI experiments into "gangs" that result in the same intermediate machine state, and only completes one of the simulations. GangES *"leverages program structure to carefully select when to compare simulations and what state to compare"* [HVAN14], and the authors report a 57.7 percent reduction in simulation time.

3.3.3 *Summary*

To summarize, FI campaigns – and, hence, the time or the required computing power to achieve test, measurement, or fine-grained analysis results – can be accelerated by either reducing the total experiment count, or by decreasing the time spent in each single FI experiment. For experiment-count reduction, several approaches from the literature leverage information from the golden run. They group equivalent experiments, and execute only a single *pilot* experiment from each group. The time spent in each experiment can be reduced by approaches aiming at the fast-forwarding phase before the FI step, or at the monitoring phase afterwards.

3.4 ALTERNATIVE ANALYSIS TECHNIQUES

As the literature has not scrutinized the potential of complete fault-space coverage for fine-grained analysis yet – not least because most FI tools do not yield information on the complete fault space, – the SIHFT community has invented other approaches aiding the configuration of SIHFT solutions. These approaches guide the placement of EDMs/ERMs either by specifically targeted FI experiments – and the application of different metrics to the results, – or by analyses completely independent of FI.

3.4.1 *Fine-Grained Analysis with Fault Injection*

Despite not having access to FI results from the complete – for example, ISA-level – fault space, parts of the literature addressed the problem of locating particularly vulnerable spots of the program by using specifically targeted FI experiments.

Hiller et al. use the PROPANE FI and error-propagation analysis framework [HJS02b] to specifically inject faults into variables grouped by their affiliation to program modules. They subsequently distill several different module-level metrics from the results – among others, the module's *exposure* to errors, its *permeability*, its *impact* on the remaining modules, and its *criticality*, – and use them for EDM placement in an aircraft arresting system [HJS01, HJS02a, HJS04]. The metrics are complicated, lack a direct relationship to the failure probability of the whole system, and are limited to module-level granularity. Similarly, Johansson et al. [JS05, Joh08] inject at the interface between an operating system and its drivers to characterize the system's capability to cope with driver failures, and define metrics like *service error permeability* and *exposure*, or *driver error diffusion*. Like PROPANE, Johansson et al.'s granularity is limited to a fixed driver-module granularity level. Leeke et al. [LJ10, LJ11] define an *importance* variable ranking for each module of the application, estimated by using FI experiments involving manual instrumentation of the target software. The approach is limited to variable-level granularity, depends on a strict modularization of the software, and only yields module-local variable importance rankings.

The fixation of these fine-grained analysis approaches to a specific granularity level – usually either a module- or variable-level granularity – already strongly predetermines and limits the analysis results, and, consequently, possible placements of EDMs/ERMs. Additionally, all mentioned approaches seem to be closely tied to a specific software structure, and partially require access to the target-software source code [HJS01, HJS02a, HJS04].

3.4.2 *Analysis Methods without Fault Injection*

To avoid the computational efforts of FI altogether, some approaches derive information on critical components of the target program by static or dynamic analyses without injecting faults. This allows to retain analysis results much faster, in some cases even enabling their use in compiler optimizations.

For example, the AVF [MWE+03a, MWE+03b] and PVF [SK09b] metrics (see Section 3.1.2.5) can be calculated using FI experiments – and are closely related to the fault-coverage factor. In a computationally cheaper approach, both Mukherjee et al. [MWE+03a, MWE+03b] and Sridharan et al. [SK09b] use a technique called *Architecturally Correct Execution* (ACE) *analysis* to approximate them without FI. Based on a single golden run of the target workload, ACE analysis pessimistically assumes *all* machine-state bits that are subsequently used (i.e., read) for computations to lead to failures. Although this approximation provides a safe upper bound for a component's failure rate – after all, only activated faults can propagate to system failures (see Section 2.2.2), – this approach has several disadvantages:

- ACE analysis cannot differentiate between different failure types, such as SDC, timeout, or reduced output quality.
- The pessimistic assumption of *all* live data corruption leading to system failures is countered by the fact that, even in not specifically hardened systems, by far not all errors propagate as failures to the next system layer, or even outside the system, but are *masked.* This has been shown to overestimate the AVF by a factor of 2 to 3 [WMP07] or even up to 7 [GEJL10]. Döbel et al. [DSE13] (work I also contributed to) show that for the PVF, similar overestimations can occur.
- Assuming the overestimation is reasonably equally distributed and ACE analysis results are nevertheless usable for EDM/ERM placement, the analysis is inherently incapable to assess a system *after* hardening it with SIHFT measures. SIHFT explicitly and by definition introduces error masking by one or more of the redundancy types listed in Section 2.4.1, and even simple error masking cannot be measured by ACE analysis.

Nevertheless, due to the low computational cost compared to FI, ACE analysis is popular in the field for assessing hardware (AVF) or software (PVF) components. In this context, Rehman et al. [RSKH11] propose the *Application Vulnerability Index* (AVI), composed of values from their *Function Vulnerability Index* (FVI), and recursively their *Instruction Vulnerability Index* (IVI), which in turn is derived in a way comparable to Mukherjee's AVF [MWE+03a, MWE+03b]. Based on these derived metrics, Rehman et al. control reliability-optimization passes in a compiler [RSKH11].

In a different approach on the source-code level, Benso et al. [BDCDN+03] define a criticality function $CF(v)$ for each variable in the target program. The metric is calculated by observing each variable's usage during a golden program run: Events like variable creation and destruction, and read and write accesses, are recorded and weighted. Benso et al. tune the weights using FI campaigns by fitting the resulting variable criticality to the FI-determined SDC rate for each workload. The most critical variables are subsequently duplicated. This computationally cheap approach only works at a fixed-size variable granularity, and the authors do not provide insights on its scalability or applicability to types of applications other than those the weights were tuned for.

3.5 SUMMARY: OBSERVATIONS AND GAPS IN THE STATE OF THE ART

This section summarizes several observations on the state of the art in FI tools and techniques, and identifies gaps this dissertation intends to fill.

3.5.1 *Fault-Space Coverage*

The first observation is that all but the most recent FI tools [DBG+09, GDJ+11, HANR12, LT13] are incapable of complete fault-space exploration. The few tools that offer this capability – partially through advanced fault-space pruning techniques [HANR12, LT13] (see Section 3.3) – advertise this as a means for full *testing* coverage on the particular machine-abstraction level they operate on. There exists no FI tool that provides advanced analysis capabilities based on complete fault-space exploration down to the level of single data objects, variables, modules, functions, source-code lines, or any other level of granularity potentially aiding fault-tolerance related design decisions.

Chapter 4: FAIL tool, and case studies leveraging full ISA-level fault-space coverage for fine-grained analysis purposes.*

To fill this gap, with the design and implementation of FAIL*, Chapter 4 provides an FI tool with full ISA-level fault-space coverage (contribution 1, cf. Section 1.2.2). The case studies in Chapter 4 take first steps in gauging the potential of full fault-space result data for fine-grained *analysis* purposes (contribution 5, cf. Section 1.2.7).

3.5.2 *Measurement and Comparison*

Another observation in the FI-tool landscape is that most existing tools primarily aim at *testing* – not only those capable of full fault-space coverage (see above). Those that provide metrics for *measurement* only measure the fault-coverage factor (see Section 3.1.2.5).

Chapter 7: Dissection of current practices in FI-result interpretation, and introduction of a novel comparison metric.

To fill this gap, Chapter 7 will show that the fault-coverage factor is unsuitable for *comparison* purposes, a property essential for driving design decisions for SIHFT. Subsequently, I will introduce an alternative metric usable for comparison purposes (contribution 4, cf. Section 1.2.6).

3.5.3 *FI-Tool Maintainability*

Most existing hardware devices or simulators do not have FI capabilities of their own. The literature provides two distinct approaches to add this capability to such execution environments. In consequence, besides the classification chosen in Section 3.2 – a breakup along the FI *technique* dimension, – FI tools can roughly be partitioned into what I call *generalists* and *specialists*: The former aim at high flexibility regarding FI target back-end – the target hardware device or simulator – and portable experiment code, the latter specialize on a single target and offer deep system-state access combined with variable fault models.

3.5.3.1 *Generalists*

The *generalists* claim a certain level of flexibility regarding the target-platform back-end. Among the benefits of this approach is that experiments can more easily be reused on a different platform – for example for gaining evidence the tested fault-tolerance measure is not platform-specific, or to move from a simulator back-end to a real hardware prototype in later development phases.

With GOOFI and GOOFI-2 [AVFK01, VAS$^+$05, SBK10], Aidemark, Vinter, Skarin et al. describe such a generic FI framework, abstracting away target systems in a plugin-based architecture. Fidalgo et al. [FGAF06] describe a generic tool addressing FI via the Nexus on-chip debugger interface. Another example is Qinject by David et al. [DCCC08], injecting faults into a target back-end utilizing the GDB debugger interface.

These approaches have the common disadvantage that the chosen interface between experiment engine and target back-end heavily limits access to target-system state, and narrows the possibilities for FI – for example obstructing the possibility to inject networking-device–specific faults into QEMU in the latter example.

3.5.3.2 *Specialists*

In contrast, the *specialist* tools are highly specific to a single target. An example is FAUmachine [PSDC07, SPS09], an x86 simulator designed from the ground up with FI in mind. FAUmachine provides access to a large part of its x86 simulator's state, and enables various FI methods, including, for example, hard-disk faults. Instead of developing an own simulator, David and Campbell [DC07] fork QEMU [Bel05] and modify its source-code to provide FI capabilities. Similarly, Parasyris et al. [PTAB14] patch the gem5 simulator [BBB$^+$11] to create their GemFI tool.

One advantage of specialists is their provision of access to the back-end's full capabilities; another one, primarily making sense in the context of hardware simulators, spending little runtime overhead for context-switching between FI and the execution environment. Unfortunately, specialist FI tools are also characterized by severe maintainability issues: Deep state-access usually results in deep intrusion into the back-end's code base. Unfortunately, enhancing a simulator with FI code implemented in a traditional imperative language such as C or C++ often leads to intermixing the implementation of different *concerns* – in this case particularly the *simulation* and *fault-injection* concerns.

Listing 3.1 illustrates this so-called *tangling* effect [KLM+97] on a code example taken from FAUmachine, where different concerns – sanity checks, normal simulator operation, and FI – are implemented in a single module dealing with IDE hard-disk simulation. A related problem, the distribution of a concern implementation across multiple implementation artifacts, is called *scattering*. The resulting *tight coupling* between simulator and FI code often makes it difficult or even impossible to exchange the tool's target back-end later on. In the case of tools that were forked from an existing hardware simulator, such as QEMU[14], even keeping in sync with the simulator's mainline evolution is often too arduous.

3.5.3.3 *Separation of Concerns*

To fill this gap, FAIL*'s implementation aims at achieving the advantages of specialists – deep device-state access, and little context-switching overhead – without paying for them with the aforementioned maintainability disadvantages. Chapter 4 will describe how FAIL* achieves this by using AOP techniques [KLM+97] in its implementation.

Chapter 4: Separation of concerns in FAIL using AOP techniques.*

3.5.4 *Back-end and FI-Technique Flexibility*

A side effect of the aforementioned maintainability problem is that most FI tools are limited to one specific simulator or hardware-device back-end. This inflexibility becomes relevant when the requirements, for example regarding the CPU architecture, change during development. In these cases, the developer most of the time must switch to a different FI tool, and start over defining the experiment procedures for the new tool.

Additionally, the FI technique is predetermined in most tools: For example, simulation-based FI tools cannot be used for FI techniques other than simulation-based FI. This forces a costly switch to a separate tool when in later development stages, FI experiments are supposed to be repeated on real prototype hardware, using, for example, TAP-based FI (see Section 3.2.2.3). The only exceptions to this general FI-technique inflexibility are GOOFI [AVFK01, VAS+05, SBK10], which supports TAP-based FI and pre-runtime and runtime SWIFI, and Xception [MCS95, CMS95, CMS98, MHCM02] and FlexFI [PRSR98, BRSR99], which both support TAP-based FI and runtime SWIFI.

Chapter 4: Execution-environment and FI-technique flexibility in FAIL.*

To fill this gap, FAIL* (Chapter 4) provides execution-environment abstractions that allow the reuse of experiment-procedure descriptions, and imple-

14 For example, David and Campbell's QEMU fork [DC07] was later seemingly given up in favor of a *generalist* [DCCC08].

```c
 1 static int
 2 ide_gen_disk_read_raw(struct cpssp *cpssp, uint8_t *buffer, uint32_t blkno)
 3 {
 4     unsigned int i;
 5     unsigned int defect;
 6 
 7     assert(blkno < cpssp->phys_linear + RESERVED_SECTORS);
 8 
 9     for (i = 0; ; i++) {
10         if (i == sizeof(cpssp->fault) / sizeof(cpssp->fault[0])) {
11             defect = 0;
12             break;
13         }
14         if (cpssp->fault[i].type == BLOCK
15          && cpssp->fault[i].blkno == blkno) {
16             defect = cpssp->fault[i].val;
17             break;
18         }
19     }
20 
21     switch (defect) {
22     case 0:
23         /* No read error. */
24         storage_read(cpssp->media, buffer, 512, blkno * 512ULL);
25         return 1;
26     case 1:
27         /* ECC corrected read error. */
28         storage_read(cpssp->media, buffer, 512, blkno * 512ULL);
29         return 0;
30     case 2:
31         /* un-correctable read error. */
32         memset(buffer, 0, 512);
33         return -1;
34     default:
35         assert(0); /* Mustn't happen. */
36     }
37 }
```

Listing 3.1: Code excerpt of FAUmachine's hard-disk simulation code (raw-block read): The implementation of *sanity check* (gray, lines 7 and 34–35), *normal simulator operation* (green, lines 23–25), and *fault injection* (white, remaining lines) concerns is heavily tangled throughout the whole code base.

ments concrete back-end connectors for several target devices and simulators, and different FI techniques.

3.5.5 *Public Availability*

A final – and in practice quite research thwarting – observation in the FI-tool landscape is that, despite the myriad of tools described in the literature, they only very rarely are publicly available. Exceptions to this general unavailability are only:

- FAUmachine [PSDC07, SPS09] – with the aforementioned maintainability and back-end fixation issues,
- the commercial, closed-source Xception tool [MCS95, CMS95, CMS98, MHCM02] – which cannot be used for research on FI itself, as its source code is not available,
- and very recent tools, such as LLFI [TP13, LFW^{+}15], GemFI [PTAB14], or GRINDER [WPS^{+}15, Win15] – which could not be used and enhanced in this thesis due to their very recent publication.

Schiffel et al. [SSSF10b] make similar observations on the low availability of FI tools – but ironically do not publish their own tool described in the very same paper [SSSF10b].

This tool scarcity in itself may be the primary reason for the existence of so many FI tools in the literature. Nevertheless, also a certain degree of "not invented here syndrome"[15] may contribute to the creation of quick in-house FI-tool hacks. To complete the circle, these quick hacks are then classified by their creators to be too specific for the target device or fault-tolerance solution they work on, or to be of too low feature completeness or source-code quality, that they are, again, not made available to the public.

To fill this gap, I made FAIL*'s source code publicly available[16] in 2014. Since then, FAIL* has been used by other researchers for research on FI itself, for the assessment of SIHFT solutions, or in teaching.

In the next chapter, I will describe the design and implementation of FAIL*, and demonstrate its measurement, comparison, and fine-grained analysis capabilities by means of several case studies.

15 In the sense of Richard Feynman's quote: *"What I cannot create, I do not understand."*
16 Online git repository: https://github.com/danceos/fail

4 FAIL*: A VERSATILE FAULT-INJECTION FRAMEWORK

"Handling errors is just attention to detail.
Injecting errors is rocket science."

— Steve Chessin, *Injecting Errors for Fun and Profit* [Che10]

Contents

This chapter describes the design and implementation of Fail*, a flexible and versatile architecture-level FI tool and framework for developers designing and deploying SIHFT measures. Fail*– short for **Fa**ult **I**njection **L**everaged[1], with the asterisk highlighting its variability regarding target back-ends – aims at filling the gaps identified in Section 3.5, and additionally provides the foundation for the remaining contributions of this dissertation in the next chapters.

After recapitulating the design goals for Fail* in Section 4.1, Section 4.2 gives an overview of the two layers Fail* is built from: the *plumbing* layer with low-level primitives and a high degree of freedom for the FI-experiment developer, and the *assessment-cycle* layer providing good defaults for widely-used ISA-level fault models, fault-space pruning, database storage, tool support, and fine-grained analyses. Sections 4.3 and 4.5 describe each layer in detail, and Sections 4.4, 4.6, 4.7, and 4.8 demonstrate their functionality in four case studies. Section 4.9 discusses the results and revisits the design goals from Section 4.1, and Section 4.10 summarizes the chapter.

Parts of this chapter were originally published on PRDC 2011 [SNK$^+$11], ARCS 2012 [SHK$^+$12], DSN 2013 [BSS13a], SOBRES 2013 [BSS13b], EDCC 2015 [SHD$^+$15], and in IJCCBS [SKSE13] and IEEE TDSC [BSS15].

4.1 DESIGN GOALS

This section recapitulates the design goals for Fail*, distilled from gaps in the state of the art that were identified in Section 3.5.

1 I only discovered long after naming Fail* that the Hoarau et al.'s FAIL-FCI [HTV07] and Joshi et al.'s PreFail [JGS11] share a quite similar name.

4.1.1 *Fault-Space Coverage*

The first design goal of Fail* is to provide the capability for full ISA-level fault-space coverage. First of all, this allows *testing* a fault-tolerance mechanism very thoroughly, catching even rare corner cases where the mechanism fails. Additionally, FI results for the complete fault-space renders novel fine-grained *analysis* methods possible that are out of reach for traditional sampling methods.

Fail* achieves complete fault-space coverage by applying advanced fault-space pruning techniques – both classic def/use pruning, as described in Section 3.3.1.1, but also a novel, heuristic pruning method described in Chapter 5 – and massive parallelization, allowing to fully exploit the power of computing clusters. The fine-grained analyses, demonstrated in the case studies throughout this chapter, go down to the level of single variables, CPU instructions, or high-level program code lines.

See Chapter 5 for details on the Fault-Similarity Pruning (FSP) heuristic.

4.1.2 *Measurement and Comparison*

The second goal is to make Fail* not only usable for testing and fine-grained analysis, but also and explicitly for *measurement* and *comparison*. Therefore, the tools described in Section 4.5 not only allow measuring the classic fault-coverage factor metric (see Section 3.1.2.5), but also *(extrapolated) absolute failure counts* as a metric more adequate for comparison in the context of SIHFT.

See Chapter 7 for details on the extrapolated absolute failure count *metric and its derivation.*

4.1.3 *Maintainable Back-End Extension*

As the third goal, Fail* aims at achieving the advantages of *specialist* FI tools (see Section 3.5.3) without inheriting the maintainability issues observed in other tools in the field. The aspired advantages are primarily the ability for deep device-state access, and little context-switching overhead between the workload execution environment and the FI extensions.

Manually patching FI capabilities into an existing simulator, like in GemFI [PTAB14], or even writing an own simulator with FI enhancements, like in FAUmachine [PSDC07, SPS09], can lead to unmaintainable code with scattered and tangled concern implementations (see Section 3.5.3). Therefore, Fail* aims at separating the concerns *simulation* and *FI* by modularizing their implementation with the help of AOP techniques [KLM+97].

4.1.4 *Back-end and FI-Technique Flexibility*

Another, central design goal of Fail* is to allow developers to profit from its testing, measurement, and fine-grained analysis capabilities without having to decide for a specific FI technique – with its advantages and disadvantages. Consequently, Fail* currently supports *two* of the FI techniques mentioned in Section 3.2 that provide the level of repeatability and controllability necessary for detailed post-injection analysis (see Table 3.3):

- *Simulation-based FI* (see Section 3.2.4) with currently three different simulator target back-ends (Lawton's Bochs [Law96], Binkert et al.'s gem5 [BBB$^+$11], and Bellard's QEMU [Bel05]) and two target architectures (x86-32 and ARM).
- A *hybrid technique* between TAP-based FI (see Section 3.2.2.3) and runtime SWIFI (see Section 3.2.3). Concretely, Fail* injects into JTAG-controlled ARM Cortex-A9 development boards, such as the PandaBoard [Pan], and improves the observability by additional SWIFI components on the target device.

Integrating these techniques in one tool allows the developer to switch back and forth from simulators to hardware during the development of a growing software system, and even to a different target architecture if the requirements change.

4.2 FAIL* OVERVIEW

In general, Fail*'s design can be divided into the *plumbing* and *assessment-cycle* layers, where the latter builds upon and abstracts from the former.

The basic *plumbing layer* provides a client/server infrastructure for parallel experiments, and abstracts from a concrete execution environment, such as a hardware simulator, or actual development hardware connected via a TAP. On the server, a user-defined *campaign* generates experiment parameters – the FARM model's fault set F (see Section 3.1.2) – and collects results from the clients. A client is essentially a hardware simulator – or a TAP-controlling program, – extended by an execution-environment abstraction layer, which in turn is controlled by a user-defined *experiment* procedure.

Using only the plumbing layer, the developer has complete freedom regarding campaign and experiment procedures. This allows the developer to work with arbitrary ISA-level fault models and workload readouts, but has the disadvantage that every detail of both campaign and experiment procedures must be implemented. Consequently, this layer is primarily aimed at researchers working on FI techniques and optimizations themselves. Section 4.3 describes the plumbing layer, and Section 4.4 demonstrates its capabilities in a case study on permanent faults in main memory and their detection by an online memory tester for Linux.

Building on the APIs provided by the plumbing layer, the *assessment-cycle layer* restricts the degrees of freedom to its users, trading them for simpler use and tool support for the complete fault-tolerance assessment cycle. This cycle spans all activities from the initial experiment definition and pre-injection/fault-space pruning steps, followed by the FI campaign itself, and ending with post-injection analyses. These analyses distill useful information from the FI results in a central database, before the assessment cycle enters its next iteration. The primary restriction deliberately introduced by the assessment-cycle layer is its focus on two widely used ISA-level fault models, namely uniformly distributed single-bit flips in main memory and CPU registers.

In contrast to the plumbing layer, the assessment-cycle layer primarily aims at the developers and users of SIHFT mechanisms. Using the features introduced by the assessment-cycle layer, this target audience can pick from ready-to-use campaign and experiment procedures, which maintain standardized fault lists and experiment outcomes in a database server. Additionally, a fault-space pruning tool helps with def/use pruning, sampling, and also a novel pruning heuristic described separately in Chapter 5. Section 4.5 describes the assessment-cycle layer, and Sections 4.6, 4.7, and 4.8 demonstrate its capabilities – also regarding fine-grained analysis possibilities emerging from the collected data – in three case studies. Two of these case studies deal with an embedded operating system protected by SIHFT methods, and the third describes a postgraduate-student level lecture at an international winter school.

4.3 PLUMBING LAYER: BACK-END ABSTRACTION AND EXTENSION

As already outlined in the overview, the *plumbing layer* provides basic services for FI experiments:

- First of all, it provides services necessary for experiment parallelization in a client/server infrastructure, including job queues – filled by a user-defined *campaign*, – and messaging primitives tailored to the requirements of the experiment at hand.
- Secondly, it abstracts from a concrete execution environment, such as a hardware simulator, or actual development hardware connected via a TAP. The client is essentially a hardware simulator (or a TAP-controlling program), extended by an *Execution-Environment Abstraction* (EEA) layer, which in turn is controlled by a user-defined *experiment* procedure.

The following sections describe FAIL*'s architecture on this layer, provide details on the implementation with a focus on the Bochs [Law96, MS08] back-end, and demonstrate its capabilities on a case study with an online memory tester for Linux [SNK+11, SKSE13].

4.3.1 *Architecture*

The plumbing layer is primarily organized in a server and a client side, as depicted in Figure 4.1.

4.3.1.1 *Server Side: Campaign and JobServer*

On the server side, the *CampaignController* manages a user-defined *Campaign*, which generates the FARM model's fault set F (see Section 3.1.2) in

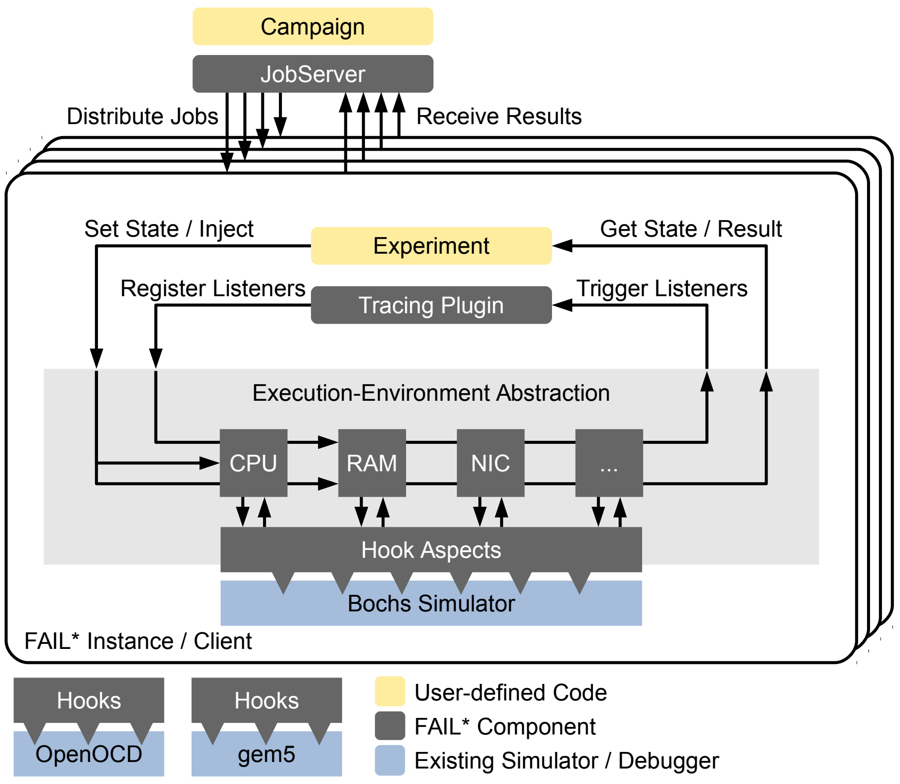


Figure 4.1: FAIL* plumbing-layer architecture overview: The *JobServer* distributes parameter sets from a user-defined *Campaign* throughout the FAIL* instances. Each FI experiment consumes a parameter set, and controls its target back-end through the *Execution-Environment Abstraction* (EEA) layer. Actual target back-ends (simulators, or real prototype hardware) can be exchanged by providing an interfacing module to this abstraction. *Adapted from [SHD+ 15].*

the form of experiment-specifically typed job parameters, encapsulated in a user-defined derivation of *ExperimentData*. These job parameters are fed into an outgoing job queue in the *JobServer*, which distributes them to clients with a *JobClient* counterpart. The *JobServer* also accepts and collects results, and hands them back to the user-defined *Campaign* to store or directly post-process them.

4.3.1.2 *Client Side*

A FAIL* *client instance* is, in principle, an extended existing simulator or debugger. Its control flow is primarily determined by a central, user-defined class derived from *ExperimentFlow*, as shown in the class diagram in Figure 4.2. This user-defined experiment procedure is started as a coroutine in parallel to the simulator's[2] workload execution. This means that either the user-defined experiment is running, or – after it has explicitly given back control – the simulator part. In this context, I chose coroutines over full-

2 The process works similarly with a debugger such as OpenOCD [Rat05], but I will focus on the extension of *simulator software* for the remainder of this section.

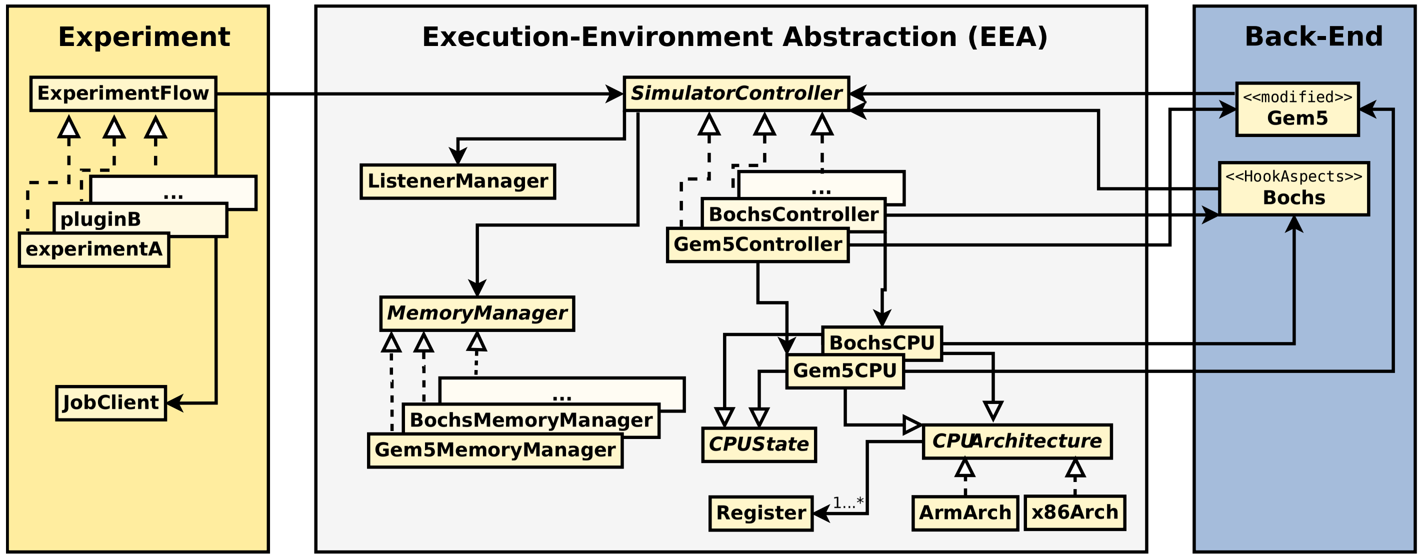


Figure 4.2: Close-up of the most important classes on the plumbing layer's client side: A user-defined experiment derived from *ExperimentFlow* controls the back-end through a *SimulatorController* specialization, supported by state and meta-information managing, architecture and back-end specific helper classes.

fledged threads because they are lightweight, require no explicit synchronization, and because real concurrency between experiment and simulator execution is not needed at all.

The *SimulatorController*[3] in the central part of Figure 4.2 is the entry point to Fail*'s key component, the *Execution-Environment Abstraction* (EEA). This component consists of additional helper classes, such as the *RegisterManager*, the *MemoryManager*, or the *ListenerManager*. It provides a common interface for the different target back-ends to the user-defined experiment procedure. Its API offers access to both target back-end meta-information and the current state, and allows registering *Listeners* for several types of events, modeled as a hierarchy of *Listener* subclasses. The Fail* API currently provides abstractions for:

- *Meta-information* on the target back-end: Number of CPUs, number of registers, platform-independent naming of special registers (program counter, stack pointer), bit widths and byte order, and memory size.
- *Fine and coarse-grained state* access: Read/write access to CPU registers and memory, injection of external interrupts, access to the back end's time; save/restore of the back-end state, and reboot.
- Listeners registrable for *events in the back end*: Reaching specific program instructions (similar to breakpoints), access to specific memory addresses, CPU exceptions, external interrupts, serial I/O, passing of specific amounts of back-end time.
- *Execution control*: The experiment can, usually after registering one or more Listeners, pass control back to the simulator coroutine, resuming workload execution.

Each target back-end primarily provides specializations to the aforementioned classes, especially the *SimulatorController* and the different manager classes. Additionally, for each machine architecture adequate meta-information classes exist.

Additionally, a target back-end may introduce interfaces to target-specific state, such as a means to manipulate a network device. Naturally, experiments using such an interface cease being portable to a different target, unless an adequate abstraction is added to the generic API.

The sequence diagram in Figure 4.3 shows a typical control flow between the user-defined experiment procedure, the *SimulatorController*, and the simulator back-end. Initially, the *SimulatorController* passes control to the user-defined experiment inheriting from the *ExperimentFlow* class. The experiment begins by restoring a previously recorded back-end state, and registers several *Listeners* with the *SimulatorController*. Once the experiment has registered a *Listener* for each event type it wants to be notified of, it tells the *SimulatorController* to resume execution of the workload in the back-end.

3 As Fail*'s development was started with extending simulators in mind, the *SimulatorController* class gained its name. It nevertheless abstracts TAP-accessed prototype hardware as well.

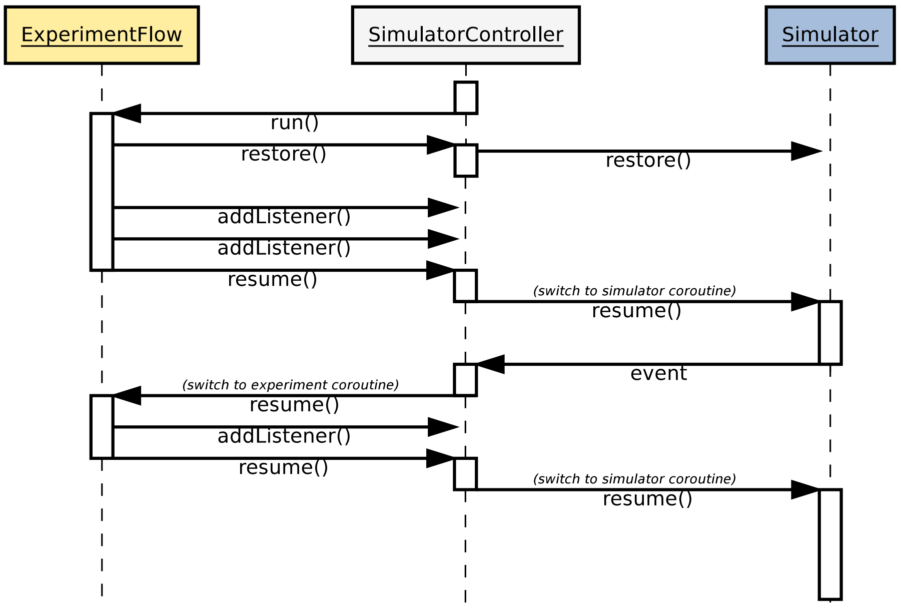


Figure 4.3: Sequence diagram of a typical control flow between the user-defined experiment procedure, the *SimulatorController*, and the simulator back-end: Controlled by the experiment, either the experiment or the simulator back-end coroutine runs exclusively.

Subsequently, the *SimulatorController* resumes the simulator by switching to its coroutine. Each time an event in the simulator matches a registered *Listener*, control is passed back to the *ExperimentFlow*.

Note that the EEA can in fact be used by more than one experiment coroutine at once. The assessment-cycle layer (Section 4.5) will use this feature by introducing reusable plug-ins, for example implementing tracing.

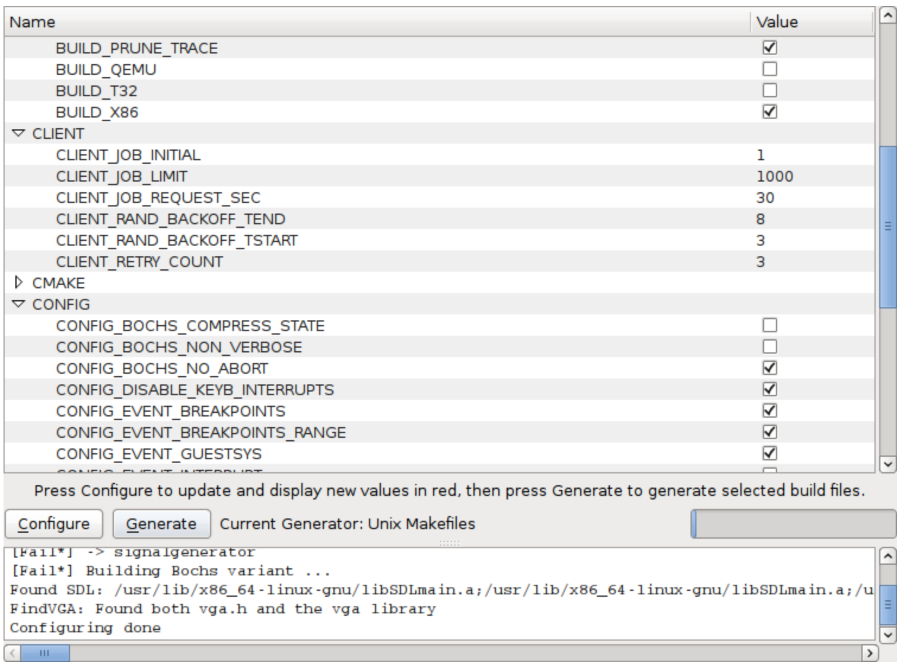
4.3.2 *Implementation*

Fail*'s plumbing layer is implemented in about 17 300 lines of C++ code[4] in the `src/core/` directory tree, although a certain assessment-cycle layer overlap with helper libraries in `src/core/util/` exists. The following sections provide details on the implementation of experiment-specifically typed job messages, and how Bochs and other back-ends are tapped into.

4.3.2.1 *Static Configuration*

By means of the *CMake* build system, several aspects of Fail* are configurable at compile-time (Figure 4.4). Primarily, the user must decide for one specific target back-end and architecture. Additionally, several details are configurable as well, for example which *Listener* types or event sources to enable, and, hence, which simulator hooks to activate; or which assessment-

4 Effective lines of code (excluding empty lines and comments), obtained with the *cloc* utility: http://cloc.sourceforge.net



Name	Value
BUILD_PRUNE_TRACE	☑
BUILD_QEMU	☐
BUILD_T32	☐
BUILD_X86	☑
▽ CLIENT	
CLIENT_JOB_INITIAL	1
CLIENT_JOB_LIMIT	1000
CLIENT_JOB_REQUEST_SEC	30
CLIENT_RAND_BACKOFF_TEND	8
CLIENT_RAND_BACKOFF_TSTART	3
CLIENT_RETRY_COUNT	3
▷ CMAKE	
▽ CONFIG	
CONFIG_BOCHS_COMPRESS_STATE	☐
CONFIG_BOCHS_NON_VERBOSE	☐
CONFIG_BOCHS_NO_ABORT	☑
CONFIG_DISABLE_KEYB_INTERRUPTS	☑
CONFIG_EVENT_BREAKPOINTS	☑
CONFIG_EVENT_BREAKPOINTS_RANGE	☑
CONFIG_EVENT_GUESTSYS	☑

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: Unix Makefiles

```
[Fail*] -> signalgenerator
[Fail*] Building Bochs variant ...
Found SDL: /usr/lib/x86_64-linux-gnu/libSDLmain.a;/usr/lib/x86_64-linux-gnu/libSDLmain.a;/u
FindVGA: Found both vga.h and the vga library
Configuring done
```

Figure 4.4: FAIL*'s *CMake*-based static configuration: The user can decide at compile time which target back-end and architecture, which Listener types or event sources, and which helper tools to compile.

cycle layer tools and plug-ins to compile, and whether to build those parts requiring specific external libraries, such as LLVM.

The configuration partially directly affects the build process, and for example compiles only the chosen simulator back-end, and partially propagates into the C++ preprocessing steps by means of generated `#define` directives.

4.3.2.2 *Communication*

As the plumbing layer is meant to provide complete freedom to the FI-experiment developer, the structure of the experiment-parameter messages exchanged between *JobServer* and *JobClient* must – like the experiment procedure itself – be defined by the developer. For this purpose, FAIL* uses the Google Protocol Buffer tools and libraries [Goo08]. They allow to define messages in a simple description language, and to generate message-type specific, lightweight [GDK11] code for serialization and de-serialization when sending messages over the network. Listing 4.4 on page 103 shows an example message type from the RAMpage case study, which is described in Section 4.4.

The communication itself is implemented as a client-initiated TCP connection, over which *JobServer* and *JobClient* exchange control messages and experiment-parameter data. The control messages themselves are also implemented using protocol buffers (Listing 4.1):

- A FAIL* client can request pilots – for performance reasons usually more than one to keep busy for a configurable amount of time – with a

```
message FailControlMessage {
    enum Command {
        // Client commands:
        NEED_WORK = 0;      // request [job_size] jobs
        RESULT_FOLLOWS = 1; // followed by [job_size] results
        // JobServer commands:
        WORK_FOLLOWS = 6;   // followed by [job_size] jobs
        COME_AGAIN = 7;     // "come back later"
        DIE = 8;            // "campaign over, terminate"
    }
    required Command command = 1;
    // ... [details omitted]
    optional uint32 job_size = 5;
}
```

Listing 4.1: Control-message type for communication between *JobClient* and *JobServer*: A FAIL* client can request several jobs at once to reduce the client-server communication overhead.

FailControlMessage with `NEED_WORK` in the `command` field. The `job_size` determines the number of requested pilots.

- Depending on the progress of the campaign, the *JobServer* replies with either `WORK_FOLLOWS` and a series of experiment-specific job-parameter messages, with `COME_AGAIN` if the job queue is currently empty but expected to fill up again, or with `DIE` if the campaign is over and the FAIL* client may terminate.
- After having obtained results from FI experiments, the client returns results with a `RESULT_FOLLOWS` message and, again, a series of experiment-specific job-parameter messages – only this time with a filled results part.

This procedure is repeated until the FI campaign is complete, and all FAIL* clients have terminated.

4.3.2.3 *Back-End Extension: Bochs*

To avoid the scattering and tangling effects described in Section 3.5.3, the Bochs [Law96, MS08] simulator back-end is extended with the help of AOP [KLM+97], concretely an AOP extension to C++ called AspectC++ [SL07]. The primary idea of AOP is to allow *strict separation of concerns* even if the underlying program structure otherwise would not allow this. So-called *aspects* – defining *where* ("pointcut") FI-specific code (*what:* "advice") should be applied – allow for compact, well-encapsulated realizations of FI concerns. An "aspect weaver" automatically takes care of compile-time intermixing of simulator and extension code.

Listing 4.2 exemplifies the approach, showing an AspectC++ implementation of an aspect diverting control flow from the memory module of the Bochs simulator to the *SimulatorController* class, completely eliminating the

```
 1 aspect MemAccess {
 2     // ...
 3     pointcut mem_write() =
 4         "void ...::bx_cpu_c::write_virtual_%(...)";
 5 
 6     advice execution (mem_write()) : after () {
 7         fail::ConcreteCPU& triggerCPU =
 8             fail::simulator.detectCPU(getCPU(tjp->that()));
 9         unsigned s = *(tjp->arg<0>()); // segment selector
10         uint32_t offset = *(tjp->arg<1>());
11         uint32_t laddr = tjp->that()->get_laddr32(s, offset);
12 
13         // divert control flow to FI module
14         fail::simulator.onMemoryAccess(&triggerCPU,
15             laddr, sizeof(*(tjp->arg<2>())), true,
16             getCPU(tjp->that())->prev_rip);
17     }
18     // ...
19 }
```

Listing 4.2: Excerpt of an AspectC++ implementation of a memory-access hook diverting control flow from the Bochs simulator into the FI extension. *Where* the aspect hooks is specified in the *pointcut* expression (green), and *what* is executed in the *advice* (gray).

need to invade the simulator's code manually. The *pointcut* expression, specifying *where* the aspect takes effect, uses wildcard expressions ("..." and "%") to match twelve different Bochs functions that implement memory writes for several address and data-operand sizes. The *advice*, denoting *what* is done "after" (line 6) each matched joinpoint, uses the join point API (see [SL07] for details) to retrieve the actual parameters of the function call within Bochs. It then collects contextual information, such as the linear address of the memory access and the CPU that triggered it, and passes it on to FAIL*'s *SimulatorController* – concretely, its `onMemoryAccess` method.

A pointcut is described in a pattern-matching language independent of the target code the advice is supposed to modify. Ideally, the target code itself – in this case, the simulator implementation – does not have to be prepared for being woven into. Additionally, one aspect – such as the `MemAccess` example in Listing 4.2 – can and often is supposed to affect more than one location of the target code. In the literature, these AOP properties are usually termed *obliviousness* and *quantification* [FF00, FECA04].

Figure 4.5 visualizes the quantification of 15 Bochs-specific hook aspects (among other functionality, hooking into Bochs' initialization code, memory accesses, interrupt and trap handling, I/O operations, branch instructions, and the instruction-executing CPU loop) in the Bochs simulator code. Each colored line represents a joinpoint where a pointcut in one of the hook aspects matches (totaling 122 source-code locations), and the weaver inserted the advice code. For example, the full version of the aforementioned `MemAccess` aspect alone – with a much longer list of pointcut expressions – is

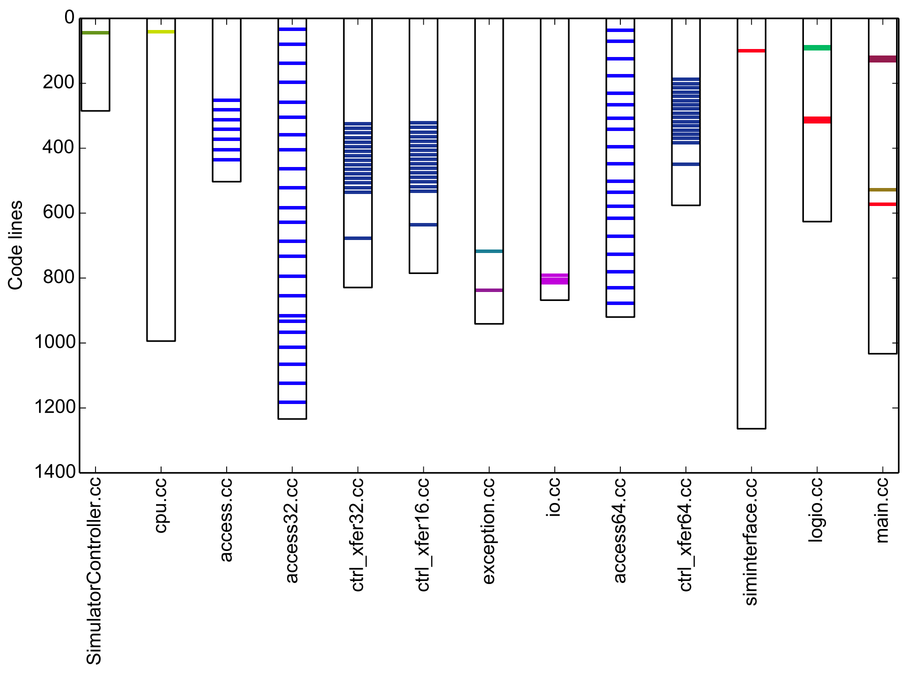


Figure 4.5: *Quantification* of 15 Bochs-specific hook aspects (variously colored): In total, the AspectC++ weaver modifies 122 code locations (`config.cc` with 3915 lines of code and five matching code locations omitted) in the Bochs source code.

responsible for a total of 49 code insertions in `access.cc`, `access32.cc`, and `access64.cc`, all colored blue. In consequence, the use of AOP strongly reduces concern scattering and tangling in the Bochs back-end, making FAIL*'s FI extension to this simulator more maintainable than a manual implementation.

Nevertheless, not all Bochs extensions were made with aspects. In total, 23 code locations in Bochs were modified manually, in most cases to add or repair functionality necessary for FI experiments. For example, the *save* and *restore* capabilities of Bochs already existed in the original code, but restore was not implemented to be triggered at runtime. Other changes add code to cope with extreme hardware-interface misuses the simulator could not deal with before – for example, sending incomprehensible command codes to the floppy-disk controller, – but can arise in rare occasions when a fault-injected target workload goes mental. Finally, one change breaks the idealized obliviousness of AOP by manually adding a "dummy" function call to the main CPU loop for aspects to hook into.

4.3.2.4 *Back-End Extension: gem5 and QEMU*

Besides Bochs, all other FAIL* back-end extensions were not implemented with AOP, for practical reasons regarding AspectC++.

In the case of the gem5 simulator, AspectC++ could not be used as gem5 makes extensive use of C++ templates. Currently, AspectC++ cannot weave into C++ template instances. Consequently, gem5 was manually patched,

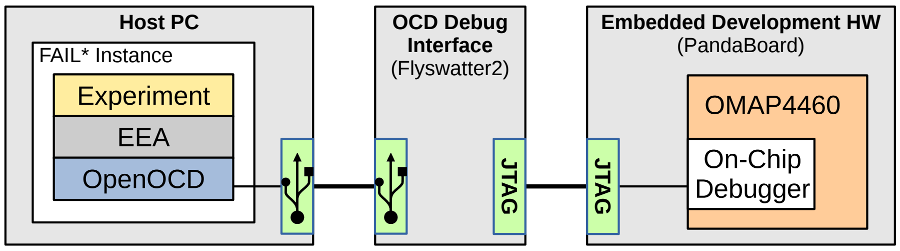


Figure 4.6: FAILPanda architecture: FAIL*'s EEA layer links into OpenOCD, which controls the On-Chip Debugger (OCD) on the PandaBoard development hardware via latency-afflicted USB and JTAG connections.

with a focus on minimal intrusion to maintain a bearable level of code maintainability.

The QEMU extension was also implemented manually: QEMU is implemented in ANSI C, which AspectC++ cannot weave into either. Also, from FAIL*'s back-end abstractions, only memory-state access and memory-access listeners were implemented for QEMU[5], as this sufficed for the injection of permanent memory errors in the RAMpage case study (see Section 4.4). In fact, the QEMU back-end is currently not planned to become feature-complete: QEMU's JIT-compilation CPU provides a high simulation speed, but strongly exacerbates the implementation of the extension hooks necessary for FAIL*.

4.3.3 *FAILPanda: OpenOCD TAP/SWIFI Back-End*

FAIL*'s OpenOCD back-end, connecting FAIL* to real prototype hardware via the JTAG TAP, has a more complex architecture than the one outlined in Section 4.3.1. This section describes the architectural differences, and provides details on the SWIFI component required for workload-behavior observability.

4.3.3.1 *Overview*

To exemplify the provision of FI capabilities to typical low-priced embedded development hardware, we chose the PandaBoard ES[6] with a widespread, ARM-based smartphone processor, the Texas Instruments OMAP4460. To realize TAP-based FI with this hardware, FAILPanda uses the OpenOCD 7.0 [Rat05] library and a USB debugger interface – concretely, the Tin Can Tools Flyswatter2 JTAG in-circuit debugger – to access the PandaBoard's JTAG interface. Figure 4.6 gives an overview on the device chain that extends FAIL*'s influence to the development hardware.

5 This feature incompleteness also is the reason that the QEMU back-end currently resides in a separate repository, and is not part of the public FAIL* release.
6 http://pandaboard.org/

4.3.3.2 *Mapping the Plumbing API to OpenOCD Primitives*

On-Chip Debuggers (OCDs) usually provide a basic set of primitives to facilitate *debugging* of programs running on embedded hardware. Typically, the target CPU can be stopped, single-stepped, resumed, or rebooted, and memory and ISA-visible CPU registers can be read and written [FGAF06]. Additionally, most OCDs provide a limited number of hardware breakpoints and watchpoints: *Breakpoints* automatically stop the CPU when a specific instruction address has been reached, and *watchpoints* do the same when the memory address they are configured to watch is – depending on the configuration – read, written, or generally accessed.

The OpenOCD [Rat05] library provides a target-device independent abstraction for these primitives, which can easily be mapped to a subset of FAIL*'s plumbing API (see Section 4.3.1):

- *Fine-grained state access* to the device's memory and CPU registers, as well as back-end reboots, can be directly passed through to OpenOCD's state-access primitives.
- *Listeners* for reaching specific program instructions or accessing memory addresses can be implemented with OpenOCD's breakpoints and watchpoints – limited by the number of available hardware breakpoints and watchpoints.
- As the coroutine separation between experiment and OpenOCD back-end does not guarantee the workload to be in a halted state when the experiment runs – as it is executed on a distant hardware device, – an event loop wrapping OpenOCD calls and callbacks to FAIL*'s *SimulatorController* must make sure only one of the two runs at once.

From the remaining plumbing-API primitives, the implementation of *save* and *restore* can be omitted by rebooting the device before each FI experiment. The *TimerListener*, activating after passing of a specified amount of back-end runtime, and often used for detecting timeouts in an experiment, can be implemented by using timers on the host PC.

4.3.3.3 *SWIFI Components for Workload Monitoring*

The FAIL* plumbing API primitives not directly mappable to OpenOCD functionality are several *Listener* types necessary for observing workload behavior in the *monitoring* phase after FI. The *TrapListener* – which is activated when the executing CPU issues a trap caused by, for example, division by zero, or the execution of an illegal op-code – does not have a counterpart in the OpenOCD API. Similarly, using the *MemAccessListener* to observe accesses to larger memory areas – often configured to activate at *any* access outside the workload's DATA or BSS sections, – cannot be mapped to hardware watchpoints, which are in the case of the OMAP4460 not only limited to four watchpoints in total, but also can only observe memory areas with a width of four bytes.

	RUNTIME (ms)	
OPERATION	avg	σ
Reboot	1032	3.43
Read CPU Register	5.980	0.0122
Write CPU Register	25.98	0.0297
Read Memory (4 B)	11.90	0.161
Write Memory (4 B)	11.93	0.0984
Single-Step	69.99	2.17

Table 4.1: Runtimes averaged over 100 measurements for some of FailPanda's plumbing-layer primitives (with standard deviation σ).

As software on the target device itself can observe these events, the consequent approach is to implement them as SWIFI components loaded by the workload's startup code itself. Although FI itself can be done via OpenOCD, modifying the workload for purposes in the context of FI qualifies this approach as SWIFI, also implying the "probe effect" disadvantages described in Section 3.2.3.

In FailPanda, the observation of traps is implemented by installing a trap handler in the modified workload startup code, and creating a breakpoint on the first instruction of this trap handler. The observation of memory accesses within larger ranges of memory are implemented by enabling the OMAP4460's MMU at startup, by reconfiguring its page tables at runtime when a big-ranged *MemAccessListener* is added or removed, and by appropriately handling – and passing on to Fail*– memory-access violation traps caused by accesses to the corresponding areas.

4.3.3.4 *Performance Measurements*

Before running complete FI campaigns with FailPanda, we conducted timing measurements on several of the basic operations described in the previous section. The results – each averaged over 100 measurements – are summarized in Table 4.1.

The first observation is that fine-grained state accesses on CPU registers and memory words are relatively fast, in the order of 5 to 30 ms. Nevertheless, these accesses are needed relatively infrequently during an FI campaign. On the other hand, a reboot – needed once at the beginning of *every* experiment – takes more than one second, which already limits the experiment throughput to below one experiment per second.

Still, the measurement with most impact on the experiment throughput is the time a *single-step* operation requires. As described in Section 3.3.2.1, fast-forwarding to a specific dynamic instruction N can in its simplest form be implemented by single-stepping. In this case, the time required for fast-forwarding grows linearly with N:

$$t_{\text{ff}} = N \times t_{\text{singlestep}}$$

With the measured single-step time of 69.99 ms, fast-forwarding the PandaBoard only to dynamic instruction $N = 400$ takes almost 28.0 seconds. For workloads with dynamic-instruction counts in the order of 10^5 to 10^9, the resulting fast-forwarding runtime clearly grows beyond feasible single-experiment runtimes.

Hence, before running actual FI campaigns with FailPanda, we had to invent a smarter fast-forwarding technique. As already announced in Section 1.2.3 and Section 3.3.2.1, Chapter 6 will fill this gap.

4.3.4 *Summary*

To summarize, Fail*'s plumbing layer provides facilities for parallelizing FI experiments through a client/server model, and abstracts from a concrete execution environment. The user-defined experiment controls and observes the workload running in the back-end by a device-independent set of primitives, which is already sufficient to implement the FI campaign for the RAMpage online memory tester in the next section.

4.4 CASE STUDY: RAMPAGE

The *RAMpage*[7] case study on permanent faults in main memory and their detection by an online memory tester for Linux directly uses Fail*'s plumbing layer. First I will outline the case study's background by describing the workload, then provide details on how the user-defined experiment and campaign implementations use Fail*'s API, and conclude with a summary of the FI-related results.

Note that in the original papers on RAMpage [SNK+11, SKSE13], we[8] also evaluated several other non-functional properties, such as memory-test speed, impact on performance and latency, and energy-consumption increase. As these contributions do not directly relate to the contributions of this dissertation, I refer the reader to the original PRDC 2011 and IJCCBS 2013 papers [SNK+11, SKSE13].

4.4.1 *Workload: An Online Memory Tester for Linux*

As described in Section 2.3, several physical phenomena can lead to permanently damaged memory cells in contemporary computing hardware. While

7 Online git repository: https://github.com/danceos/rampage

8 As described in Section 1.4, RAMpage itself, its detailed evaluation, and the resulting publications [SNK+11, SKSE13], were a collaborative effort. Whenever in this dissertation I refer to results obtained together with other researchers, I switch from the first person singular to plural.

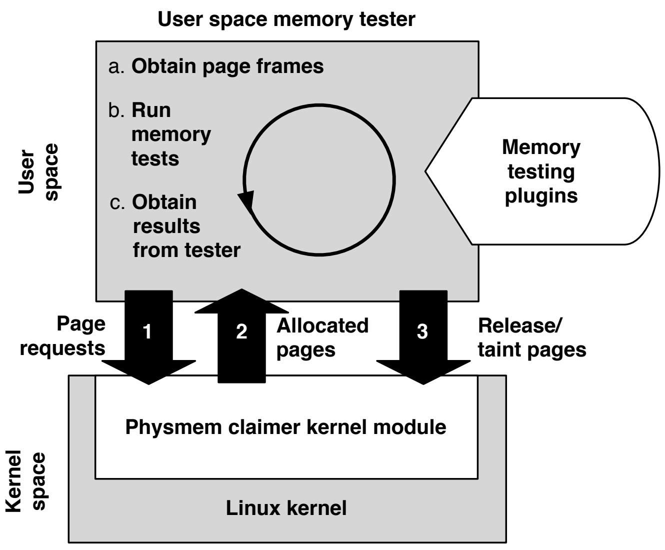


Figure 4.7: Overview on RAMpage's architecture and basic operation: A user-space memory tester obtains physical page frames from the *Physmem-claimer* kernel module, and runs memory tests on them. Depending on the test result, the page frames are returned to the kernel for normal use, or flagged/tainted to contain faulty bits.

high-end systems usually employ ECC or other hardware-based error correction methods (see Section 2.4.2), these are prohibitively expensive for small-scale, cost-sensitive systems. However, reducing the probability of undetected permanent memory errors is also a worthwhile goal for these systems. Systematic memory tests are a useful tool to detect errors: If these tests are performed with a sufficiently high frequency, most errors should be discoverable.

However, shutting down a server system in order to perform tests, for example using software like Memtest86+ [Dem11], is in many cases unacceptable. As a consequence, we developed RAMpage, an approach to run a memory tester as a system service during normal operation. Compared to existing memory-test mechanisms, RAMpage requires no downtime and allows for flexible selection of memory test frequency and test methods, and works mostly as a user-space process.

Figure 4.7 gives an overview on RAMpage's architecture and basic operation. A user-space memory tester obtains physical page frames by requesting them from the second RAMpage component, the *Physmem-claimer* kernel module. The module tries – depending on the claimer configuration – several methods to claim the requested frames from the kernel, and maps those it succeeded claiming into the user-space tester's address space. The user-space program in turn runs plugin-provided memory tests – here we ported many of the testing algorithms Memtest86+ [Dem11] offers – on the returned page frames. Depending on the test result, the page frames are returned to the kernel for normal use, or flagged to contain faulty bits and removed from the kernel's page-frame allocation pool.

4.4.2 *Fail* Experiment Setup*

The main purpose of conducting FI experiments with FAIL* was to test RAMpage's effectiveness – or, in other words, to prove that the online test can in fact detect permanent memory errors.

As QEMU's workload-execution speed outperforms that of Bochs, and RAMpage needs to be run as part of a full-fledged Linux environment, we chose FAILQEMU – a FAIL* configuration with the QEMU back-end enabled – as the evaluation vehicle. The workload includes a Debian Linux 6.0 installation with a 64-bit x86-64 SMP-enabled Linux 3.5 kernel, with RAMpage configured to start claiming page frames and testing memory right after Linux has completed the boot process. RAMpage's diagnostic output – including messages indicating the detection of a memory error – was configured to be output on a serial interface in order to be observable by the experiment. Note that for the FI experiments, we set up RAMpage to run in an otherwise idle system, in contrast to the setup with additionally running benchmark programs for the efficiency measurements in the original publications [SNK$^+$11, SKSE13].

The FI experiments are supposed to systematically inject permanent single-bit faults in the simulated machine's main memory, and to observe whether RAMpage correctly detects the fault. In terms of dividing this procedure into campaign and experiment,

- the *campaign* produces the fault-set F, each fault being represented by its memory address and bit position, and fault type[9], and
- the *experiment* implements the procedure of injecting *one* fault, and is parametrized by one $f \in F$.

The code excerpt in Listing 4.3 illustrates the *campaign* implementation for RAMpage. The loop starting at line 5 systematically walks the single-bit permanent-fault space in a "top-down" fashion: By bit-reversing the iteration variable `n`, the resulting addresses start out spanning large parts of the 2047 MiB simulated memory – represented in the `address_bits` variable containing the number of bits a valid memory address can have, – and successively fill the spaces in-between the already visited addresses.[10] The loop body then instantiates a *RAMpageExperimentData* container for each address, fills it with concrete job parameters (lines 8 to 11), and pushes it into the outgoing job queue (line 13). Notice that in this campaign, only stuck-at-one faults (line 11) were tested. Finally, the campaign collects incoming result data, and writes them to a file for subsequent analysis (lines 17 to 22).

9 Permanent memory faults can manifest, for example, in the form of bits stuck at zero or one, or more complex patterns, such as direct or inverse coupling with other bits [DBT90].

10 Concretely, first the addresses 0 and 1024 MiB are visited, then 512 MiB and 1536 MiB, followed by 256/1280/768/1792 MiB, and so on. This pattern makes sure that the complete memory is visited in a more and more fine-grained raster, allowing to terminate the campaign early if the planned experimentation time runs out. If the campaign is allowed to run to completion, the final raster width is 8192 bytes, or two 4 KiB pages.

```
bool RAMpageCampaign::run() {
    // ... [initialization omitted]

    // systematically march through the fault space
    for (uint64_t n = 0; n < 1024*256; ++n) {
        uint64_t addr = reverse_bits(n) >> (64 - address_bits);

        RAMpageExperimentData *d = new RAMpageExperimentData;
        d->msg.set_mem_addr(addr);
        d->msg.set_mem_bit(4);
        d->msg.set_errortype(d->msg.ERROR_STUCK_AT_1);
        // ... [more parameter details omitted]
        campaignmanager.addParam(d);
    }
    campaignmanager.noMoreParameters();

    // collect results
    RAMpageExperimentData *res;
    while ((res = static_cast<RAMpageExperimentData *>
                     (campaignmanager.getDone()))) {
        // ... [output to CSV table omitted]
    }

    // ... [cleanup omitted]
}
```

Listing 4.3: Excerpt of the *campaign* implementation for testing the RAMpage online memory tester: The code systematically walks the address space down to a granularity of two 4 KiB page frames, generates and distributes job parameters, and collects incoming results.

Listing 4.4 shows the description of the Google Protocol-Buffers job-message type, which is encapsulated in *RAMpageExperimentData* containers in both campaign and experiment. The message is reused for both distributing jobs and returning experiment outcomes, and is consequently divided into an input part (green in Listing 4.4) the experiment is parametrized with, and an output part (gray) returned to the campaign. Among other details, the input comprises the injection address and bit number (`mem_addr` and `mem_bit`), and the fault type to inject (`errortype`). Besides omitted details, the output is one of several possible `ResultTypes` specifying RAMpage's successful error detection, or one of several possible failures.

The general *experiment* procedure behind injecting permanent faults into memory via FAIL*'s plumbing API is the following:

1. Register a *MemWriteListener* for the faulty memory address, which will activate when the workload writes data to this address.

2. Resume the simulator and wait for the listener to activate.

3. Each time this happens, read the newly written data byte from the specified memory address with the *MemoryManager*, modify the affected bit or bits, and write the byte back to memory.

```
message RAMpageProtoMsg {
// Input: experiment parameters
    // memory error address, and failing bit at that address
    required uint64 mem_addr = 1; required uint32 mem_bit = 2;
    // coupled bit at that address (only for ERROR_COUPLING)
    optional uint32 mem_coupled_bit = 3;
    // error type
    enum ErrorType { ERROR_NONE = 1;
        ERROR_STUCK_AT_0 = 2; ERROR_STUCK_AT_1 = 3;
        ERROR_COUPLING = 4; ERROR_INVERSE_COUPLING = 5;
    }
    required ErrorType errortype = 4;
    // ...

// Output: experiment results
    enum ResultType {
        RIGHT_PFN_DETECTED = 1; // successful detection
        WRONG_PFN_DETECTED = 2; // detected in wrong page frame
        PFN_WAS_LISTED = 3; // not detected although PF was tested
        NO_PFNS_TESTED = 4; // no PFs tested over longer periods
        LOCAL_TIMEOUT = 5; GLOBAL_TIMEOUT = 6; UNKNOWN = 7;
    }
    optional ResultType resulttype = 9;
    // ...
}
```

Listing 4.4: Protocol-buffers message description for the RAMpage FI campaign: Job parameters, primarily the address of the faulty bit and the error type to inject (green), and fields for storing each experiment's outcome (gray).

From the workload's point of view, this fault re-writing procedure is indistinguishable from "real" permanent faults: The next read to the memory address will yield corrupted data.

Listing 4.5 shows an excerpt of the corresponding experiment implementation using Fail*'s plumbing API. Fail* starts an experiment as a separate coroutine with its own control flow in "parallel" to the simulator (see Section 4.3.1) by running its `run()` function (line 1) – which every experiment inheriting from `ExperimentFlow` must implement.

The first action is to ask the *JobServer* – which was fed with *RAMpageExperimentData* jobs by the campaign (see above) – for work, which is stored in the `m_param` variable (lines 2 to 5). Subsequently, several *Listeners* are instantiated (lines 8 to 11) and registered with the *SimulatorController* (lines 12 to 15): Besides the aforementionend *MemWriteListener* (line 8) that is required for injecting the permanent fault, an *IOPortListener* – triggering whenever the workload prints something on the serial port – helps capturing the diagnostic output of RAMpage's user-space component. Additionally, two *TimerListeners* (lines 10 and 11) allow to monitor different timeouts regarding passed simulation time, which is necessary for aborting the experiment when no progress has been made for too long.

```
bool RAMpageExperiment::run() {
    m_param = new RAMpageExperimentData;
    if (!m_jobclient.getParam(*m_param)) { // ask server for work
        simulator.terminate(1);
    }

    // instantiate and register listeners
    MemWriteListener l_mem1(m_param->msg.mem_addr());
    IOPortListener l_io(0x2f8, true); // ttyS1 aka COM2
    TimerListener l_timeout_local(m_param->msg.local_timeout());
    TimerListener l_timeout_global(m_param->msg.global_timeout());
    simulator.addListener(&l_mem1);
    simulator.addListener(&l_io);
    simulator.addListener(&l_timeout_local);
    simulator.addListener(&l_timeout_global);

    while (true) {
        BaseListener *l = simulator.resume(); // resume simulation
        if (l == &l_mem1) { // memory write -> re-write faulty bits
            simulator.addListener(&l_mem1);
            handleMemWrite(l_mem1.getTriggerAddress());
        } else if (l == &l_io) { // serial I/O
            simulator.addListener(&l_io);
            // check for memory-tester status output
            handleIO(l_io.getData());
        } else if (...) // ... [timeout handling omitted]
    }
}

void RAMpageExperiment::handleMemWrite(address_t addr) {
    unsigned bit1 = m_param->msg.mem_bit();
    unsigned bit2 = m_param->msg.mem_coupled_bit();
    // read byte at addr from memory via MemoryManager
    unsigned char data = m_mm.getByte(addr);
    // inject error depending on job-message's errortype
    switch (m_param->msg.errortype()) {
    case RAMpageProtoMsg::ERROR_NONE: break;
    case RAMpageProtoMsg::ERROR_STUCK_AT_0:
        data &= ~(1 << bit1); break; // stuck-at-0
    case RAMpageProtoMsg::ERROR_STUCK_AT_1:
        data |= 1 << bit1; break; // stuck-at-1
    case RAMpageProtoMsg::ERROR_COUPLING:
        data &= ~(1 << bit2); // coupling bit2 := bit1
        data |= ((data & (1 << bit1)) != 0) << bit2; break;
    case RAMpageProtoMsg::ERROR_INVERSE_COUPLING:
        data &= ~(1 << bit2); // coupling bit2 := !bit1
        data |= ((data & (1 << bit1)) == 0) << bit2; break;
    }
    m_mm.setByte(addr, data); // write back byte via MemoryManager
}
```

Listing 4.5: Excerpt of the *experiment* implementation for the RAMpage campaign: The `RAMpageExperiment::run()` function registers several *Listeners* for observing the workload's behavior, and reacts on their activation in the main event loop.

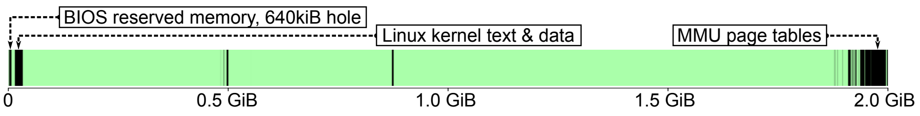


Figure 4.8: Single-bit stuck-at memory errors detected by RAMpage in FI experiments conducted with FAILQEMU: Only 5.3 percent of the faults injected at every second 4 KiB page boundary were not located (black areas) due to unsuccessful page claiming within the allotted timeouts.

In the main event loop (spanning lines 17 to 27), the experiment primarily passes control back to the simulator, resuming the workload's execution (line 18), which is in a halted state anytime the experiment code runs. The `simulator.resume()` call returns once a registered *Listener* has been activated, and returns a pointer to the *Listener* in question to the caller. The `if` cascade in lines 19 to 26 subsequently checks which of the previously registered *Listeners* was activated and caused the interruption. Each of the branches re-registers the – instantly unregistered – Listener, and passes activation details – such as the memory address that was written in case of the *MemWriteListener*, or the data that was sent to the I/O port in case of the *IOPortListener* – to a subfunction of the RAMpage experiment.

If the *MemWriteListener* was activated, the `handleMemWrite()` function follows the procedure outlined above: The new contents of the faulty byte in memory are read using the *MemoryManager* in `m_mm` – a previously initialized member of the `RAMpageExperiment` class – in line 34, modified according to the error type specified in the job parameters (lines 36 to 48), and written back (line 49).

The omitted `handleIO()` function maintains a buffer with recent workload output, and scans it for RAMpage's diagnostic messages. If the diagnostics in fact indicate that RAMpage has detected the injected fault, it records the time since experiment start, returns the job message with an adequate `resulttype` (see Listing 4.4) to the *JobServer*, and terminates FAILQEMU.

4.4.3 *Evaluation*

For a thorough test of RAMpage, and to obtain a broad data basis on which memory areas it can claim and test in a running Linux environment, we deployed an armada of FAILQEMU clients on TU Dortmund's *LiDOng* cluster, and ran the RAMpage campaign to completion.

In 94.7 percent of the 262 016 experiments, RAMpage succeeded in claiming the faulty page frame, testing the memory and detecting the fault. In 4.5 percent of the experiments, the injected fault was not detected because the corresponding page frame could not be claimed by RAMpage's kernel module: These memory areas, plotted in Figure 4.8, primarily host the kernel image or its data structures, most prominently the MMU page tables, or x86 legacy such as the "640 KiB hole". In 0.2 percent of the experiments, RAMpage's diagnostic output did not show any progress after injecting the fault – possibly a symptom of the faulty memory address being vital for the

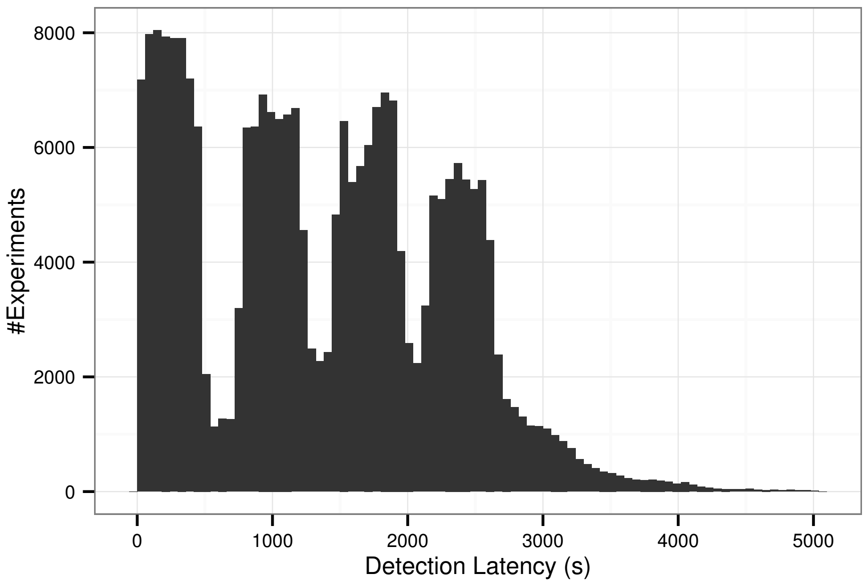


Figure 4.9: Distribution of RAMpage's fault-detection latencies (bin size: one minute), measured from the point in time the permanent-fault injection starts: In 96.5 percent of experiments with a successful detection, RAMpage detects the permanent fault in under 50 minutes.

correct operation of Linux or RAMpage itself. In the remaining 0.6 percent of the experiments, the timeout condition aborted the experiment after not having detected the fault for 120 minutes of simulator time.

Digging deeper in the result data collected by the RAMpage campaign, Figure 4.9 plots the fault-detection *latency* distribution, measured from the point in time the permanent-fault injection started. The results show that in 96.5 percent of experiments with a successful detection, RAMpage detected the permanent fault in under 50 minutes.[11]

As a general conclusion from the Fail* FI tests, RAMpage's fault-detection mechanism has a fault coverage close to 100 percent, and detection latencies sufficient to be an alternative to and outperforming offline tests in many concerns. RAMpage is only impeded by Linux's memory management not allowing to claim certain areas of physical memory, which may only be possible to improve by overhauling the Linux's memory-management code itself.

A conclusion for Fail*'s plumbing API is that the provided primitives allow to effectively implement permanent memory faults, to observe all workload readouts relevant for the presented RAMpage use case, and to parallelize large amounts of experiments to provide results timely.

4.4.4 *Reality Check: Tests on Real Hardware*

Additionally to the systematic FI-based tests with Fail*, we wanted to prove RAMpage's usability with *real* broken memory hardware. We conducted ex-

11 We did not further investigate the root cause for the four discrete peaks in the histogram at 200 s, 1000 s, 1800 s, and 2400 s.

periments on a two-way AMD Opteron 250 server with eight 1 GiB DDR1-400 modules (Apacer ECC Registered DDR-400, with ECC disabled in the BIOS), one of which was known defective. Memtest86+ runs confirmed a vast amount of single- and multi-bit errors throughout the module's complete address range.

RAMpage successfully detected all defective page frames from the faulty DIMM, and removed them from further use. After offlining the problematic address range, the machine continued running smoothly with the remaining 7 GiB of memory. Placing the defective DIMM in the "wrong" memory slot on the mainboard led to Linux failing to boot – an effect expected when essential kernel data structures or machine instructions are placed in the affected address range.

In essence, RAMpage also works correctly on real defective hardware, providing evidence that FailQEMU tested the right properties.

4.4.5 *Summary*

To summarize, Fail*'s plumbing-layer facilities sufficed to implement the FI-campaign and experiment procedures for the RAMpage memory tester. The campaign systematically walks the single-bit permanent-fault space, distributes FI jobs – encapsulated in an experiment-specific message type – via the CampaignManager, and collects and stores incoming results. Using the back-end independent EEA primitives, the experiment observes the workload running in the QEMU back-end, and injects permanent faults by rewriting the job-specific memory address after it has changed.

However, for Fail* users it can quickly become tedious to repeatedly implement the same wide-spread fault model, and to cope with multiple different benchmarks in one campaign and orders of magnitude more experiment results than in the RAMpage example. Consequently, the next section describes Fail*'s *assessment-cycle layer*, which builds on the primitives provided by the plumbing layer. It simplifies many common tasks by providing libraries and a reusable campaign, modularizes often-needed experiment components, uses a database for job and result data, and provides tools accompanying the complete assessment cycle.

4.5 ASSESSMENT-CYCLE LAYER: FAULT-MODEL DEFAULTS, PRUNING, AND TOOL SUPPORT

Fail*'s plumbing layer already provides all functionality to construct complex, target back-end independent FI campaigns, and to execute large numbers of experiments in parallel. Nevertheless, for a SIHFT developer the offered degrees of freedom induce repeated manual effort, and require expert knowledge on fault models, subtle details of the plumbing API, and result evaluation.

This disadvantage is addressed by Fail*'s *assessment-cycle layer*. From the perspective of a SIHFT developer, FI is usually – analogous to common prac-

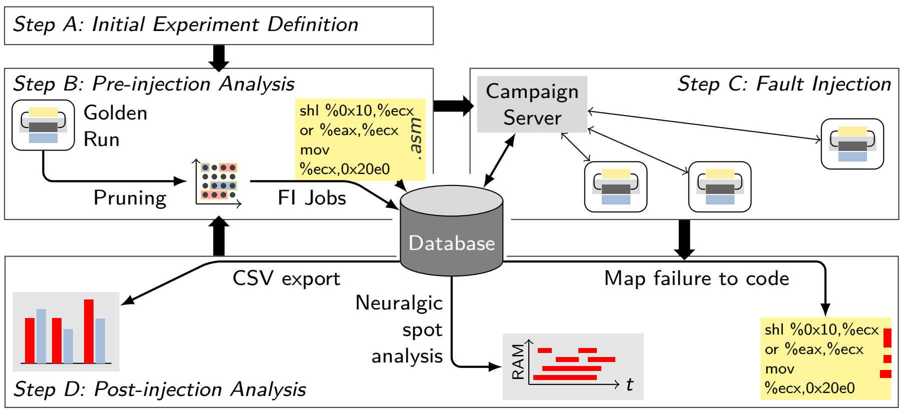


Figure 4.10: The fault-tolerance assessment cycle: After an initial *experiment definition* step, the SIHFT developer enters one or multiple iterations of the fault-tolerance assessment cycle, improving the system's fault resilience in each iteration. *Adapted from [SHD⁺ 15].*

tices in iterative software development – part of a development *cycle*: The soft-error analysis and hardening process is iterative, and supported by a *continuous fault-tolerance assessment* process. This process keeps developers informed about the robustness of their software, and repeatedly tests or measures the effectiveness and efficiency of their hardening measures.

Consequently, FAIL*'s assessment-cycle layer reduces the degrees of freedom offered by the plumbing layer by restricting the user to the classic ISA-level fault models described in Section 3.1.3: Single-bit flips in main memory or CPU registers, uniformly distributed over time. For these fault models, the layer provides ready-to-use campaign and experiment procedures, and tools supporting all steps in the fault-tolerance assessment cycle. These components and tools are implemented in about 15 400 lines of C++, Python, PHP, and shell-script code.

The following sections describe the fault-tolerance assessment cycle, and how FAIL*'s different assessment-cycle layer components support each step. After this abstract description, Sections 4.6 to 4.8 present three case studies demonstrating the assessment-cycle layer's capabilities in practical examples.

4.5.1 *The Fault-Tolerance Assessment Cycle*

Figure 4.10 shows the four principal steps in the fault-tolerance assessment process. Before entering the assessment cycle, the SIHFT developer specifies the FI-experiment procedure – tailored to the actual workloads – in the initial *experiment definition* (step A in Figure 4.10). This step is not fundamentally different from experiment definition on the plumbing layer (see Section 4.3.1.2 and Section 4.4.2), but supported and simplified by a set of pre-defined components.

```
message Trace_Event {
    required uint64 ip = 1;     // instruction pointer
    optional int64 time_delta = 6; // sim. time since previous event
    optional uint64 memaddr = 2;// memory access: address
    optional uint32 width = 3;  // memory access: width in bytes
    enum AccessType {           // memory access: read/write
        READ = 1;
        WRITE = 2;
    }
    optional AccessType accesstype = 4;
    // ... [optional extensions omitted]
}
```

Listing 4.6: Excerpt of the protocol-buffers message description for recording instruction and memory-access trace events.

Then, the developer enters one or multiple iterations of the assessment cycle, improving the system's fault resilience in each iteration:

- The *pre-injection analysis* (step B in Figure 4.10) primarily executes the golden run – a workload run without injecting faults – and records the reference readouts and a memory-access and instruction trace (see Section 3.1.2.2). The traces are subsequently used for fault-space pruning (see Section 3.3.1), resulting in the fault list F – or, in other words, a set of FI jobs that is stored in a database.
- The *fault-injection campaign* (step C) distributes the jobs to FAIL* client instances, and collects results in the database.
- The *post-injection analysis* (step D) distills useful information from the collected results. It helps the SIHFT developer by combining test, measurement/comparison, or fine-grained analysis results with details of the workload, aiding SIHFT placement and design decisions.

The following sections provide details on steps A–D, and how FAIL*'s assessment-cycle layer supports and implements them.

4.5.2 *Experiment Definition*

Similar to a developer using the plumbing layer, a SIHFT developer must provide a user-defined campaign and experiment implementation. Nevertheless, several pre-defined components simplify this procedure significantly on the assessment-cycle layer.

4.5.2.1 *Golden-Run Mode, Experiment Plugins, and Generic-Tracing*

First of all, many experiment implementations provide a *golden run mode* in which the FAIL* client does not request job parameters from the *JobServer*, and does not inject a fault. In this mode, the experiment only creates a loadable machine state every FI experiment can load to skip the boot process, and records the reference output and an instruction and memory-access trace of the workload. The assessment-cycle layer provides two plugins that modularize this functionality:

- The *Tracing* plugin records the workload's instruction and memory-access trace in a file. While the plugin is activated, it uses *Listener*s to be notified of instruction executions or memory accesses by the *SimulatorController* (see Section 4.3.1.2). For each of these events, it serializes a Fail*-wide standardized, parametrized *Trace_Event* protocol-buffer message (Listing 4.6) into a compressed trace file, which can be used later to construct the fault list.

- Similarly, the *SerialOutput* plugin can be used to record workload output: Listening to data-output events on a specified output port of the target device, it records all data bytes written while the plugin is active.

As described in Section 4.3.1.2, plugins use the same EEA API as experiments, with the sole difference that the SimulatorController does not automatically run plugins at startup. Instead, the experiment instantiates and activates a plugin, which then runs as another coroutine on par with the experiment.

In many cases, the developer does not need any additional functionality in golden-run mode besides booting the target device, saving the machine state, and recording the reference output and a trace. This functionality is bundled in the *Generic-Tracing* experiment, which uses the *Tracing* and *SerialOutput* plugins and is ready-to-use for the pre-injection analysis step.

4.5.2.2 *Experiment Definition, Database-Experiment, and Generic-Experiment*

The definition of the FI-experiment procedure itself is also simplified by plugins. For example, instead of manually adding output-recording code to the experiment's main event loop – as in the RAMpage experiment (Listing 4.5, lines 22 ff.), – the *SerialOutput* plugin helps modularizing this functionality.

Simplifying the experiment-definition step even further, the *Database-Experiment* – belonging to and inheriting its name from the *Database-Campaign*, which is described in the next section – imposes a pre-defined structure on the experiment. The developer derives the user-defined experiment from the `DatabaseExperiment` class and only implements a few workload-specific call-back functions, while the *Database-Experiment* executes the remaining, regularly needed steps:

1. Retrieve job parameters from the *JobServer*.

2. Load the initial workload state, which was previously saved in the golden-run phase.

3. Fast-forward the target workload to the injection time specified in the job parameters.

4. Inject a single-bit flip at the memory location specified in the job parameters.
5. Resume and monitor the workload until it finishes regularly, or terminates abnormally.
6. Return outcome to the *JobServer*.

Between each of these steps, a user-defined callback can introduce additional functionality, like adding a workload-specific *Listener* that observes the behavior of an important global variable in the monitoring phase.

In case only a minimal, standard set of readouts is needed, the developer can instead decide to use the *Generic-Experiment*, which concretizes the *Database-Experiment* by monitoring timeouts, CPU exceptions/traps, memory writes to the workload's text segment or other areas not belonging to the workload's memory image, and reaching a set of instruction addresses that denote the workload's termination.

4.5.2.3 *Database-Campaign*

As the Database-Experiment's counterpart on the server side, the *Database-Campaign* provides a generic implementation for the user-defined campaign. Although it plays well with both the Database-Experiment and the Generic-Experiment, it can as well be used together with a compatible user-defined experiment on the client side.

Like with the *Database-Experiment*, the `DatabaseCampaign` class must be derived from and concretized for a particular user-defined campaign. The provided base-class functionality distributes jobs from the database (see next section), collects results from the clients, and again stores them in a result table in the central database. The primary restriction for the developer is that the experiment-parameter part of the protocol-buffers message used for communicating job parameters – the green "input" half of RAMpage's example message in Listing 4.4 – is predefined in a separate *DatabaseCampaignMessage* definition. This definition must be imported into the message definition of the user's derived campaign with the still user-defined experiment-result part. In essence, this user-defined outcome message part is the only customization the developer must make.

4.5.3 *Pre-Injection Analysis*

Once the campaign and experiment procedure, and the protocol-buffer message type used for communication between the two, are defined, the SIHFT developer enters the fault-tolerance assessment cycle starting with the *pre-injection analysis* step (see Figure 4.10). This step comprises all preparations for the FI campaign: The golden run, and the synthesis of the list of FI jobs (the fault list F, see Section 3.1.2).

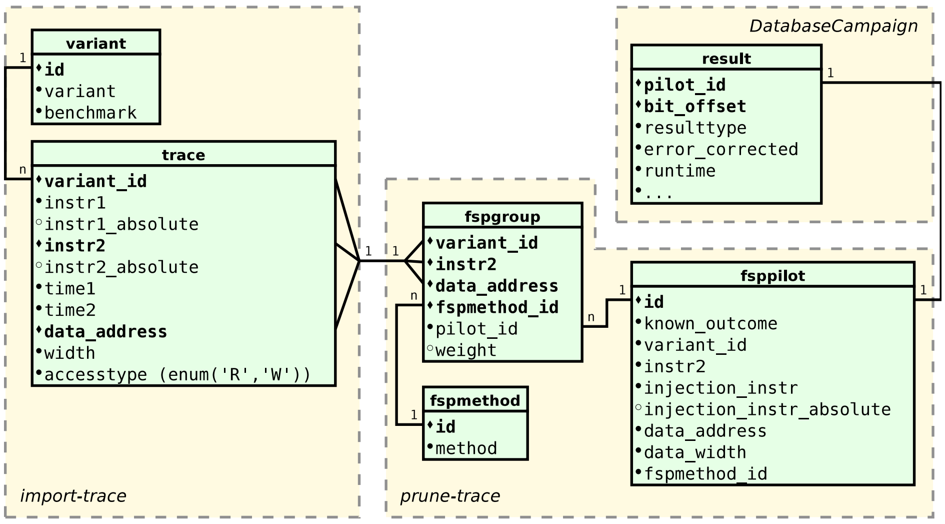


Figure 4.11: FAIL*'s basic database schema, holding information on the user's different workload *variants*, the *trace* recorded during the golden run, the FI jobs and their mapping to the workload's fault space (*fsppilot*, *fspgroup*, *fspmethod*), and the *results* obtained during the FI campaign.
Legend:
◆ = Primary-key column (additionally denoted in boldface),
● = non-nullable column (value mandatory),
○ = nullable column (value optional),
slanted = FAIL* component creating/filling these tables.

4.5.3.1 *Golden Run*

As described in the previous section, an experiment's golden-run mode – or, alternatively, FAIL*'s *Generic-Tracing* experiment – records a loadable starting point at the beginning of the target workload, a reference output for comparison during the FI campaign, and an instruction and memory-access trace. During pre-injection analysis, the developer executes the golden run for each of a possibly larger set of different workloads or workload variants, for example a *baseline* and a SIHFT-protected variant.

While the resulting loadable machine states and reference-output files are needed later during every experiment run of the FI campaign, the recorded trace is directly used as a basis for defining the list of FI jobs by means of fault-space pruning techniques such as def/use pruning (see Section 3.3.1.1), or sampling.

4.5.3.2 *Interjection: Database Support*

In order to keep the amounts of data manageable that accumulate during the fault-tolerance assessment cycle, FAIL*'s assessment-cycle layer uses a relational database for data handling. Additionally, this approach will simplify data analysis in the post-injection steps (see Section 4.5.5). Figure 4.11 shows FAIL*'s basic database schema, omitting extensions for fine-grained post-injection analyses (Section 4.5.5) and the fault-similarity pruning heuristic presented in Chapter 5. The schema consists of the following tables:

VARIANT: Holds user-specifiable names for workloads, for example benchmarks in different protection variants.

TRACE: Holds information the def/use equivalence classes directly derived from the memory-access – or, alternatively, CPU-register access – trace recorded during the golden run for each *variant*. The set of all equivalence classes for a variant represents its complete single-bit fault space.

FSPMETHOD: Holds information which fault-space pruning ("fsp") method was used for generating the fault list.

FSPPILOT: Holds the list of *pilots* – the job parameters for the FI experiments that effectively will be executed. The term "pilot" stems from terminology used in other fault-space pruning tools (see Section 3.3.1.2).

FSPGROUP: Holds a mapping which *pilot* represents what parts of the fault space in the *trace* table.

RESULT: Holds the results obtained from each FI-experiment run during the campaign.

Where applicable, the following sections will provide more context and details on each table. Currently, FAIL*'s database back-end is implemented using Oracle's *MySQL* database, respectively its community-developed *MariaDB* fork.

4.5.3.3 *Trace Import*

After the golden run, each recorded trace is imported into the *trace* database table, referring to the corresponding workload's name and variant in the *variant* table. As FAIL* uses def/use pruning as its primary experiment-count reduction method, the recorded trace is represented by its def/use equivalence classes (see Section 3.3.1.1) in the *trace* table: Each equivalence class is defined by its start and end time (`time1` and `time2` columns in the *trace* table in Figure 4.11), its memory address (`data_address`) and the `width` in bytes of the memory access it ends with, and the type of the access (read or write, `accesstype`). The remaining columns denote the dynamic-instruction count at the beginning and end of the equivalence class (`instr1`, `instr2`), and for internal sanity checks the static instruction address at these points in time (`instr1_absolute`, `instr2_absolute`).

If the SIHFT developer intends to inject memory faults corresponding to the uniform model discussed in Section 3.1.3.1, the recorded memory-access trace from the golden run can be imported directly into the *trace* table. In this case, each memory-access event in the trace maps to one[12] record in the trace table.

12 In fact, a multiple-byte access is split into multiple single-byte accesses in the current implementation to simplify further processing.

In case of a CPU-register FI campaign – with uniformly distributed single-bit flips in the CPU's ISA-visible registers, as described in Section 3.1.3.3, – the trace must be preprocessed first. Taking the instruction-trace portion of the recorded trace and the ELF binary of the corresponding workload, Fail*'s trace-import facility decodes each static instruction that was executed during the golden run, and statically determines the input and output registers it uses. An internal mapping then translates register numbers into pseudo addresses that fit into the `data_address` column of the *trace* table, generating def/use equivalence classes similar to those used in the memory fault-model case.

In Fail*'s assessment-cycle layer, the trace import is implemented in the `import-trace` tool using the `--importer MemoryImporter` and `--importer RegisterImporter` parameters, respectively. The instruction decoding necessary for the `RegisterImporter` is implemented using the LLVM [LA04] libraries.

4.5.3.4 *Fault-Space Pruning and Sampling*

In the final step of the pre-injection analysis, the SIHFT developer generates the list of pilots – the FI experiments that are stored in the *fsppilot* table and effectively executed during the following campaign. Additionally, the *fspgroup* table is filled with a mapping from each pilot to a subset of the fault space in the *trace* table: Depending on the chosen fault-space pruning method, a pilot may represent multiple def/use equivalence classes.

If the developer intends to cover the complete single-bit fault space, the pilot-list generation is straightforward. For each def/use equivalence ending with a *read* access to memory – or a CPU register, – one pilot is added to the *fsppilot* table, and noted with a one-to-one mapping in the *fspgroup* table. Additionally, for each workload variant *one* equivalence class ending with a *write* gets picked and added as a pilot: Although the result for all eight single-bit experiments is in principle known in advance to be "No Effect" (see Section 3.3.1.1), actually injecting one pilot simplifies the problem of generically finding appropriate values for the experiment-specific *result* table columns. As a side effect, these experiments also serve as a sanity check that the experiment correctly detects the workload's benign outcome. In the *fspgroup* table, this single "No Effect" pilot maps to *all write* def/use equivalence classes belonging to this workload variant.

Chapter 7 – more specifically, Pitfall 2 in Section 7.2.5, – sheds more light on biased sampling from a def/use-pruned fault space.

If the developer only needs a global measurement estimate, *sampling* the fault space suffices. In order not to bias the sample, Fail* does not randomly pick from the set of def/use equivalence classes in the trace table: As they can vary dramatically in size, their probability of being hit by a single-bit flip is also quite different. Instead, in each sampling step Fail* randomly selects a fault-space coordinate from the basic, not yet pruned fault space. Then, it adds the def/use equivalence class that contains this coordinate to both *fsppilot* and *fspgroup* tables, unless it has been selected earlier already. The `weight` column in *fspgroup* counts how often the particular equivalence class has been selected, which becomes relevant in the calculation of the estimate later.

In Fail*'s assessment-cycle layer, pruning and sampling is implemented in the `prune-trace` tool using the `--prune-method BasicPruner` repectively `--prune-method SamplingPruner` parameters.

4.5.4 *Fault-Injection Campaign*

After the pre-injection analysis step has prepared a loadable state, preserving the back-end's state right at the beginning of the workload's execution, the correct golden-run readout of the workload, and the fault list represented by pilots in the *fsppilot* table, the FI campaign can commence. The *Database-Campaign*, parametrized with a user-defined and campaign-specific protocol-buffer message type (see Section 4.5.2.3), instantiates a *JobServer* and feeds it with the pilots from the *fsppilot* database table. The *JobServer* subsequently distributes pilots to clients and collects returning results using the mechanism described in Section 4.3.2.2.

The structure of the *result* table in the database – primarily, its column names and types – is directly determined by the user-defined outcome part of the protocol-buffer message. Using a helper library named *DatabaseProtobufAdapter*, the *Database-Campaign* consequently generates this table structure from the protocol-buffer message description itself. This ensures that experiment results fit into the database upon returning from the clients.

4.5.5 *Post-Injection Analysis*

After collecting FI results for every pilot in the *result* table, Fail*'s *post-injection analysis* step helps the developer by pointing to *test* cases the SIHFT solution missed, by *measuring* the software fault-tolerance effectiveness, and by aiding the process of system hardening and the placement of SIHFT mechanisms by *fine-grained analysis.*

The basis for any of Fail*'s post-injection analyses is the raw result data collected in the *result* table. Assuming the decision for complete fault-space coverage in the pre-injection phase (Section 4.5.3.4), the *fspgroup* table provides a mapping for every def/use equivalence class in the *trace* table, and consequently for every single fault-space coordinate. Based on this raw result projection to the fault space, Fail* can derive several metrics, analyses, and visualizations.

4.5.5.1 *Global Metrics*

One extreme is to calculate a single aggregation over the complete result set of each workload variant – a *global* metric condensing the results to a single value (or a small number of values).

As described in Section 3.1.2.5, the primary global metric calculated by any FI tool is the *fault-coverage factor.* After deciding which experiment outcomes – usually stored in the *result* table's `resulttype` column – are to be interpreted as "covered" faults, Fail*'s assessment-cycle layer allows to retrieve the fault coverage with a simple SQL statement joining the *trace*, *fsp-group*, and *result* tables. In the simplest case, an SDC outcome is considered "uncovered", and all remaining results "covered". Alternatively, Fail* can calculate the frequency percentages for every existing outcome in relation to the complete fault-space size.

Chapter 7 explains in detail why the fault-coverage factor metric is problematic when comparing workload variants.

Nevertheless, as Chapter 7 will show that the fault-coverage factor is of little use for comparing different workload variants, Fail* also generates *absolute failure counts* as a global metric. In principle, this metric corresponds to the fault-coverage factor, but without the division by the fault-space size.

Both metrics – the fault-coverage factor as well as the absolute failure-count metric – can as well be derived from sampling results. In this case, Fail* provides SQL statements that also quantify the margin of error corresponding to a specified confidence level (see Section 3.1.2.5).

All three case studies in Section 4.6, Section 4.7, and Section 4.8 provide examples for using Fail*'s assessment-cycle layer to measure absolute failure counts.

4.5.5.2 *Symbol-Level Analyses*

The primary advantage arising from detailed results in every corner of the fault space – instead of a few hundred or thousand randomly distributed, punctual samples – is that result aggregation becomes possible in much smaller granules, which can be explored and defined after the campaign has already finished. The table of global symbols in an ELF binary provides a first and relatively simple means for aggregating FI results on a granularity that helps a SIHFT developer pinpoint particularly critical components in the workload.

An ELF symbol table contains memory addresses, sizes, and names of global variables, including larger areas such as thread stacks or the BSS or data segments themselves, but also of functions in the code segment. As Fail*'s assessment-cycle layer holds all result data in a relational database, importing the ELF symbols into an additional *symbol* table allows to retrieve per-symbol statistics by an additional SQL join: A trivial modification to the SQL statements for the global metrics from Section 4.5.5.1 aggregates results along the time axis for each symbol.

All three case studies in Section 4.6, Section 4.7, and Section 4.8 provide examples for symbol-level analyses.

4.5.5.3 *Leveraging Debug Information*

Beyond the ELF symbol table, Fail* can also use additional information from the workload's compilation process for detailed analyses. Running a modern compiler such as *GCC* or *clang* with the `-g` command-line parameter embeds additional information intended for simplifying debugging in the resulting ELF binary. This debug information – usually encoded according to the DWARF standard [DWA10] – provides structural information on the program, such as compilation units, functions, source-code lines, complex data structures, global and local variables, and a mapping to their representation in the generated machine code.

Similar to the symbol-table case above, FAIL*'s `import-trace` tool is also capable of importing a subset of the DWARF debug information in an ELF binary into the database, adding several tables to the schema introduced in Section 4.5.3.2:

DBG_FILENAME: Holds names of compilation units that were linked into the ELF binary.

DBG_SOURCE: Holds high-level source-code lines.

DBG_MAPPING: Provides a mapping between static instruction addresses and high-level source-code lines, allowing the attribution of failures to lines of code.

Additionally, as this is required for some of the fine-grained analyses presented in the following case studies, the tool also imports the recorded instruction trace (*fulltrace* table), and the disassembled mnemonics of the target program's code sections in the *objdump* table.

The debug information, along with function symbols, the instruction trace, and the program's disassembly, enables several fine-grained FI-result analyses. Among others, the following case studies demonstrate FI-result aggregates and criticality rankings per static machine-code instruction, per high-level source-code line, per function, or per translation unit.[13]

The WSOS case study (see Section 4.8) provides several examples for fine-grained analyses leveraging debug information.

4.5.6 *Summary*

To summarize, FAIL*'s assessment-cycle layer strongly increases ease of use compared to the plumbing layer, trading it for a restricted set of basic fault models. The developer benefits from ready-to-use campaign and experiment procedures, database support, and tools supporting all steps in the fault-tolerance assessment cycle – including novel, fine-grained analysis methods in the post-injection step.

The following sections will demonstrate these analysis methods, and the remaining assessment-cycle layer features, in three case studies.

13 As a result of Richard Hellwig's master's thesis [Hel14], even more fine-grained analyses on the level of temporary data structures – local variables, and architecture-specific return addresses and frame pointers – stored on the program stack(s) are possible. However, in the current state these analyses are too fragile under compiler optimizations to be presented as usable FAIL* features.

4.6 CASE STUDY: GENERIC OBJECT PROTECTION

The *Generic Object Protection* (GOP) is a memory-error detection and recovery SIHFT approach developed by Christoph Borchert [BSS13a, BSS15] in the context of the DanceOS project [SKLS11]. It exploits application knowledge about memory accesses, which are analyzed at compile time, and hardens the accessed data structures by information redundancy (see Section 2.4.3.4) and strategically placed, compiler-generated runtime checks. In this case study, I empirically investigated the GOP's effectiveness by running large-scale FI campaigns on a set of GOP-hardened benchmarks running on eCos [Mas02], an embedded operating system written in object-oriented C++. I used FAIL*'s assessment-cycle layer for large-scale FI campaigns, and both global and symbol-level post-injection analyses.

The following sections provide details on the first assessment-cycle iteration with a baseline assessment, and briefly outline the basic approach behind the GOP. After a first evaluation round seeking out a good GOP configuration for each benchmark, the second round provides FI measurements for the effectiveness of different GOP variants.

Note that the original papers on the GOP [BSS13a, BSS15] provide many additional details regarding the GOP's design and implementation, necessary extensions to AspectC++, a wait-free synchronization approach dealing with concurrency in the eCos kernel, and efficiency measurements. As these contributions are not my own, and are not necessary to demonstrate FAIL*'s capabilities, I omit them here and refer the reader to the original publications [BSS13a, BSS15].

4.6.1 *Fault and Failure Model*

For the GOP case study, I assumed uniformly distributed, single-bit faults in main memory (see Section 3.1.3.1). I also assumed that read-only data and code (text) is stored in far more reliable EEPROM or Flash, or is covered by other EDMs/ERMs, such as software-implemented ECC [SSM00] (see Section 2.4.3.4).

Regarding the failure model, in this case study I distinguish between "No Effect" outcomes, SDC failures, timeouts, and CPU exceptions. I consider timeouts to be malign, as they are comparable to SDCs when no hardware watchdog timer is present. Even *with* a watchdog timer, a timeout condition incurs long detection latencies that, depending on the scenario, may be undesirable. Similarly, CPU exceptions are considered malign because they are sometimes hard to distinguish from regular workload behavior. CPU exceptions are not always suitable as error detectors, and do not help with error correction.

BENCHMARK	DESCRIPTION	THR.	SYS. CALLS
BIN_SEM1	Binary semaphore functionality	2	28
BIN_SEM2	Dining philosophers	15	659
BIN_SEM3	Binary semaphore timeout	2	28
CNT_SEM1	Counting semaphore functionality	2	35
EXCEPT1	Exception functionality	1	52
FLAG1	Flag functionality	3	78
KILL	Thread `kill()` and `reinitialize()`	3	23
MBOX1	Message box functionality	2	94
MQUEUE1	Message queues	2	73
MUTEX1	Basic mutex functionality	3	40
MUTEX2	Mutex release functionality	4	46
MUTEX3	Mutex priority inheritance	7	55 308 146
RELEASE	Thread `release()`	2	117
SCHED1	Basic scheduler functions	2	18
SYNC2	Different locking mechanisms	4	2425
SYNC3	Priorities and priority inheritance	3	39
THREAD0	Thread constructors/destructors	1	4
THREAD1	Basic thread functions	2	17
THREAD2	Scheduler and thread priorities	3	41

Table 4.2: eCos kernel-test benchmarks, each testing a different subset of the kernel functionality: The last two columns show the number of threads (THR.) the benchmark runs, and the number of system calls (SYS. CALLS) invoked at runtime.

4.6.2 *Workload and Experiment Setup*

The GOP focuses on hardening operating-system kernel state, as data corruption within the kernel data structures can affect all running applications. Assuming a special-purpose embedded-systems context, the GOP conceptually exploits knowledge on the application's behavior – especially its operating-system usage profile. The working hypothesis behind the GOP is that only a small, application-specific subset of the kernel state is critical and needs protection.

As a diverse set of applications that thoroughly exercise various different functionality areas of the eCos kernel, the GOP's developer applied the mechanism to kernel-test programs bundled with eCos itself. Table 4.2 lists the 19 different benchmarks with a short description which kernel functionality they test, the number of threads they spawn, and the number of system calls they invoke at runtime. As an input for the FI-based effectiveness evaluation, the GOP developer compiled all benchmarks for IA-32 with the GNU C++ compiler (GCC Debian 4.7.2-5), and set up eCos 3.0 with its default configuration, including GCC optimization level `-O2`, and GRUB startup.

In the experiment-definition step of Fail*'s assessment cycle (see Section 4.5.2.3), I used the pre-defined *Database-Campaign* on the server side. On the client side, I used FailBochs with a custom experiment procedure that injects one transient single-bit flip per run, and afterwards records several different readouts, including traps, timeouts, test-program outputs, and special GOP signals indicating the detection or correction of an error.

4.6.3 *Baseline Assessment*

As a first assessment step, I ran the pre-injection analysis and FI-campaign steps for the unmodified *baseline* variant of the workloads listed in Table 4.2, provided by the GOP's developer in the form of ELF binaries. To get an overview of the – application-specifically – most critical eCos components, I then imported the ELF symbol tables of each binary, and generated per-symbol aggregations (see Section 4.5.5).

Table 4.3 exemplarily lists the top ten symbols that caused the THREAD1 and MUTEX1 benchmarks to fail – either with an SDC, a CPU exception, or a timeout, which are subsumed in the #FAILURES column. For the THREAD1 benchmark, the top ten symbols amount to 97.8 % of all observed abnormal program terminations, with the global `thread` variable – an array of `Cyg_Thread` data structures – being the most critical. The MUTEX1 results display similar data-structure names on the top end of the symbol ranking, yet with a different ordering: Its focus on a different area of kernel functionality puts the `stack` data structure, containing all thread stacks, in front of `thread_obj`, as it exhibits a different operating-system usage profile.

In general, the baseline assessment revealed that in the chosen set of workloads, the stacks and the global kernel data structures are the most suscep-

	SYMBOL	SIZE	#FAILURES	%
THREAD1	`thread`	264	9.20×10^9	39.5 %
	`Cyg_RealTimeClock::rtc`	52	4.29×10^9	18.4 %
	`stack`	5088	3.31×10^9	14.2 %
	`Cyg_Scheduler::scheduler`	132	8.53×10^8	3.7 %
	`comm_channels`	96	8.53×10^8	3.7 %
	`pt1`	4	8.53×10^8	3.7 %
	`Cyg_Interrupt::dsr_list_tail`	4	8.53×10^8	3.7 %
	`hal_interrupt_objects`	896	8.53×10^8	3.7 %
	`hal_interrupt_handlers`	896	8.53×10^8	3.7 %
	`Cyg_Scheduler_SchedLock::sched_lock`	4	8.53×10^8	3.7 %
MUTEX1	`stack`	7632	1.69×10^6	31.8 %
	`thread_obj`	416	1.49×10^6	28.0 %
	`cvar2`	12	2.88×10^5	5.4 %
	`cvar1`	8	2.26×10^5	4.6 %
	`m0`	20	2.08×10^5	3.9 %
	`comm_channels`	96	1.95×10^5	3.7 %
	`m1`	20	1.84×10^5	3.5 %
	`cvar0`	8	1.78×10^5	3.3 %
	`Cyg_Interrupt::dsr_list`	4	1.56×10^5	2.9 %
	`Cyg_Scheduler::scheduler`	132	1.49×10^5	2.8 %

Table 4.3: FI results from the GOP baseline assessment: Top ten fault-susceptible symbols (or, contiguous memory areas) for the unmodified THREAD1 and MUTEX1 benchmarks. "Failure" counts subsume SDC, CPU exception, and timeout results.

GOP VARIANT	DESCRIPTION
Baseline	Unmodified benchmark, GOP disabled (no redundancy).
CRC	A CRC-32 implementation (EDM).
TMR	Triple-modular information redundancy, using two copies of each data member and majority voting (EDM/ERM).
CRC+DMR	CRC (EDM, see above), plus one copy of each data member for additional error correction (ERM).
SUM+DMR	A simple 32-bit checksum (EDM), plus one copy of each data member (ERM).
Hamming	Hamming code (EDM/ERM).

Table 4.4: GOP variants and a short description which information-redundancy method they implement.

tible. This analysis motivated the GOP developer to primarily target those "critical spots".

4.6.4 *Generic Object Protection*

As already mentioned in the beginning of this case study, the GOP exploits static application knowledge about memory accesses. This knowledge is extracted from the target application assuming an object-oriented programming model: The GOP introduces information redundancy on a granularity of C++ classes. The remaining problem is *when to check* – and potentially repair – consistency between the data and its redundancy, and *when to update* the redundancy after the data was written to.

In case of the GOP, the developer solved this problem by exploiting the object-oriented program structure of the target software, which associates sequences of data reads and writes with method calls: Without direct data-member accesses, methods are the only pieces of code reading and writing data members of a class. Consequently, the GOP places redundancy checks and updates around method-function calls:

- *Before* a method of a critical object is executed, the mechanism checks whether the object suffered from a memory fault.
- *After* the execution of the method, redundancy for the object's state is stored.

Similar to the way FAIL* taps into the C++ code of the Bochs simulator back-end (see Section 4.3.2.3), the GOP's developer used AspectC++ to modularize the check and update functionality, and to extend a configurable list of target-software classes with data members holding the necessary redundancy. Concretely, he implemented several GOP variants reimplementing different classic information-redundancy techniques. Table 4.4 lists each variant with a short description.

4.6.5 *Evaluation: Finding an Optimal GOP Configuration*

Besides the decision whether error detection (EDM) suffices or correction is required (ERM), and the selection of a concrete variant from Table 4.4, the GOP's primary degree of freedom is the configuration which target-software classes to protect, and which to leave unprotected.

Although enabling the protection for *all* target-system classes may seem to be the optimal configuration, each inserted redundancy check and update increases the workload's execution runtime – and, hence, its "attack surface" to transient faults. Consequently, the GOP developer decided to explore the trade-off between the subset of selected classes and the runtime overhead caused by the EDMs/ERMs. In a first step, he provided a set of benchmark variants with *increasingly more classes* being hardened with the CRC variant of the GOP.

Figure 4.12 shows the results of an extensive FI campaign with the FAIL* setup described in Section 4.6.2, totaling 87 million single FI experiments. The absolute failure counts – extrapolated from sampling estimates for the BIN_SEM2, FLAG1, KILL and SYNC2 benchmarks – are broken down into SDCs, timeouts, and CPU exceptions. The results indicate that the benchmarks can be differentiated into three classes:

- The first benchmark class exhibits lower failure counts the more eCos-kernel classes are CRC-protected with the GOP, for example BIN_SEM2 or THREAD1.
- The second class of benchmarks increases in failure-count numbers for any GOP configuration, for example BIN_SEM1 or SYNC2. The fault susceptibility of these programs actually *worsens* by applying the GOP, because the additional attack surface from the runtime and memory overhead outweighs all gains from being protected.
- The third class of benchmarks – for example EXCEPT1 – lies somewhere in-between the other two classes: The failure count *decreases* for the first few but then *increases* with more CRC-protected classes.

The second benchmark class simply cannot be protected by the GOP: The SIHFT-induced additional runtime and memory-usage increases the chance of being hit by uniformly-distributed memory bit flips beyond the point where the increased attack surface is not compensated by the fault-tolerance gains anymore. For the remaining benchmarks, the GOP developer picked the optimally-protected configuration, and also devised a heuristic that helps choosing a near-optimal configuration even without FI experiments [BSS15].

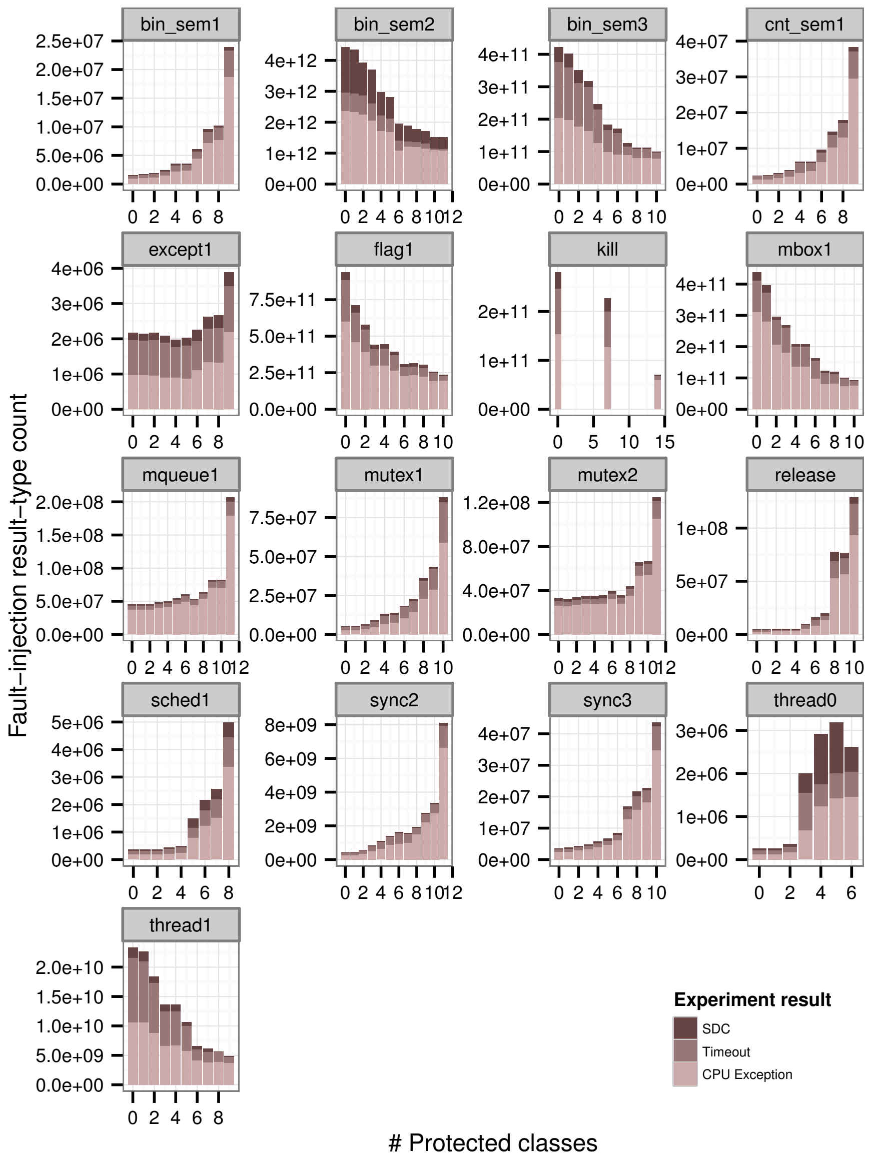


Figure 4.12: FI results for increasingly more GOP-protected classes (actual class names omitted, baseline corresponds to zero protected classes): Some benchmarks *improve* in all GOP configurations (e.g., BIN_SEM2 or THREAD1), some *worsen* in all variants (e.g., BIN_SEM1 or SYNC2), and some have a failure-count minimum with a small set of protected classes (e.g., EXCEPT1). Results for BIN_SEM2, FLAG1, KILL and SYNC2 are sampling estimates with a maximum relative standard error of 3.55 %. Due to their long runtime, I only measured three configurations for KILL, and omitted the MUTEX3 and THREAD2 benchmarks.

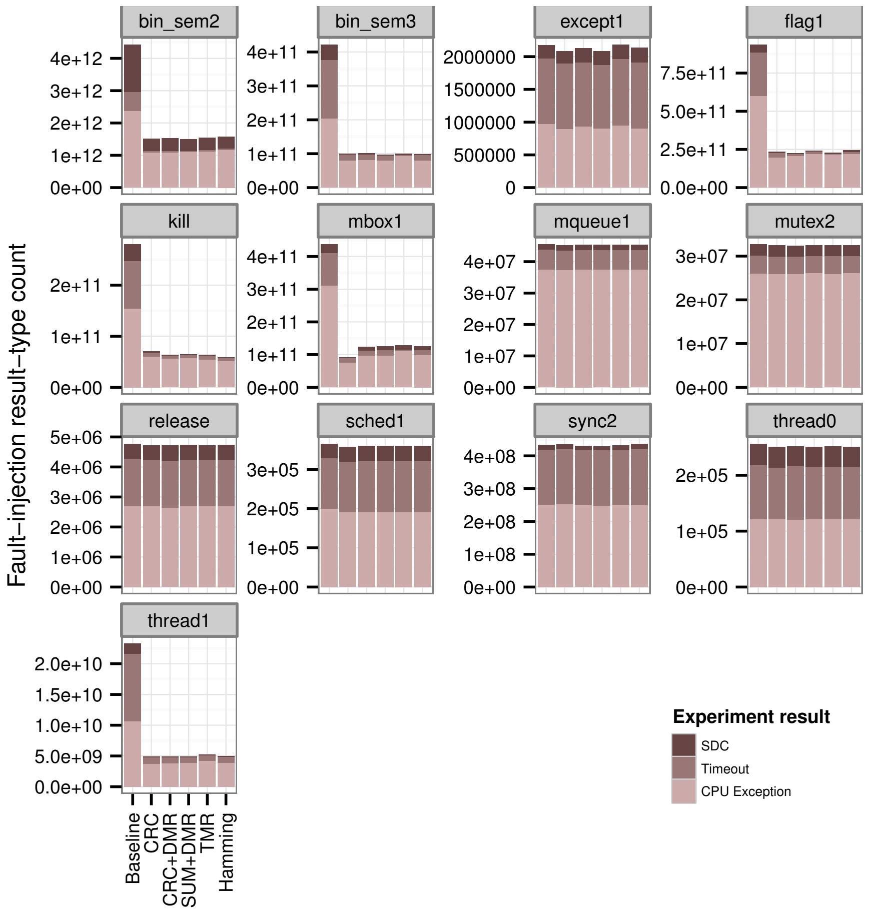


Figure 4.13: Protection effectiveness for different EDM/ERM variants. Benchmarks BIN_SEM1, CNT_SEM1, MUTEX1, and SYNC3 are missing, as the analysis in Section 4.6.5 showed that the GOP does improve the baseline in any configuration. Results for BIN_SEM2, FLAG1, KILL and SYNC2 are sampling estimates with a maximum relative standard error of 4.24 %.

4.6.6 *Evaluation: Variant Comparison*

Using the same optimized configurations as for the CRC variant from the previous section, the GOP developer provided benchmark binaries protected by the remaining GOP EDMs/ERMs variants described in Table 4.4. Figure 4.13 shows the FI results that compare the different mechanisms among each other, obtained in an FI campaign amounting to additional 46 million experiments. The results indicate that the five EDMs/ERMs are similarly effective in reducing failure counts, and – in the case of MBOX1 and THREAD1, protected with CRC – reduce the failure probability by up to 79 percent compared to the baseline. The benchmarks with little improvement in the optimal case in Figure 4.12 do not significantly improve with the other ERMs, either. The total number of system failures – compared to the baseline without GOP – is reduced by 69.14 percent in the CRC variant, and, for example, by 68.75 percent in the CRC+DMR error-correction variant.

For the FI experiments without sampling – all benchmarks except for BIN_SEM2, FLAG1, KILL and SYNC2, – even 74.11 percent of the baseline's failures are prevented by the Hamming variant. Only 1.54 percent of the failures are not covered by the GOP and still originate from protected kernel data structures, whereas most remaining failures (24.35 percent) stem from unprotected data, such as the stacks of the application and the kernel.

4.6.7 *Conclusions on Fail* Tooling*

In GOP case study, I successfully applied a wide range of Fail*'s assessment-cycle layer features and capabilities, as summarized in the following sections.

4.6.7.1 *Experiment Definition*

In the experiment-definition step (see Section 4.5.2), I used the pre-defined Database-Campaign on the server side, and a custom experiment procedure controlling the Bochs simulator via the Fail* plumbing layer's EEA API (see Section 4.3.1.2). Additionally to the capability of injecting single-bit flips – that would as well have been provided by the *Generic-Experiment*, – the custom implementation allowed to record several GOP-specific readouts.[14]

4.6.7.2 *Importing and Pruning*

Fail*'s `import-trace` and `prune-trace` tools helped importing all required data to a local MariaDB 10.1 server, as described in Section 4.5.3.3 and Section 4.5.3.4. I primarily used def/use pruning for experiment-count reduction to enable fine-grained analyses in the post-injection phase, but resorted to sampling for the overly large BIN_SEM2, FLAG1, KILL, and SYNC2 benchmarks. In total, def/use pruning respectively sampling reduced the raw fault-space size[15] from 2.13×10^{15} (respectively 1.16×10^{13} if cycles spent in CPU wait-states are not counted) to 1.33×10^{8}.

4.6.7.3 *Fault-Injection Campaign*

For the CRC-focused GOP-configuration optimization (see Section 4.6.5) and the final evaluation of the different EDM/ERM variants (see Section 4.6.6), I ran the resulting 133 million single experiments in parallel on TU Dortmund's *LiDOng* cluster. The parallelization reduced the total simulation runtime of 2.5 CPU years to a few days of shared cluster usage.

Figure 4.14 plots a histogram of FI-experiment wall-clock runtimes: On average, each experiment ran for 0.58 s, but a long outlier trail reaches up to

14 In fact, the first versions of the GOP-specific custom experiment-procedure definition predate the Generic-Experiment by about two years. With marginal additions, the *Generic-Experiment* would be equally usable for the GOP case study if it were repeated with the latest Fail* release, entirely eliminating the need for a custom experiment implementation.

15 Used memory bits × CPU cycles from start to end.

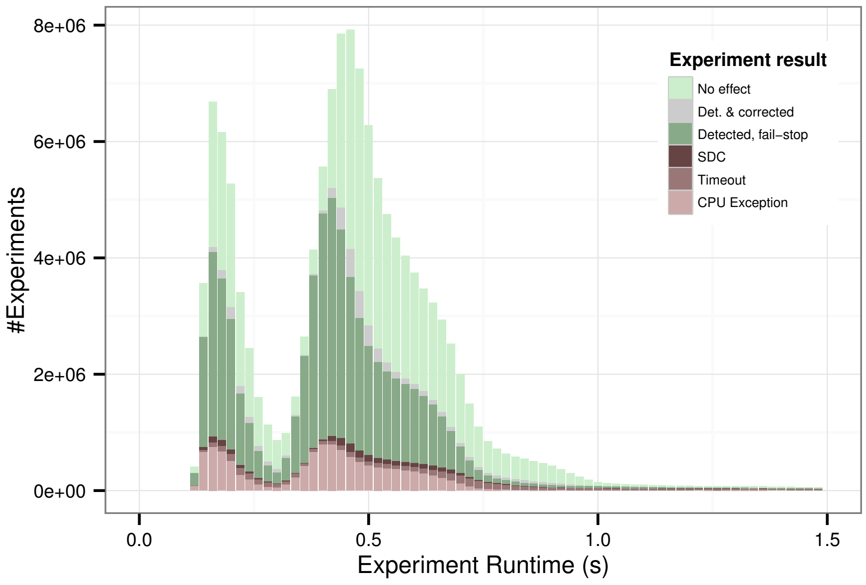


Figure 4.14: Wall-clock FI-experiment runtime histogram for the GOP campaign: At an average runtime of 0.58 s, the two data-point accumulations around 0.2 s and 0.5 s are primarily owed to the host-CPU speed diversity in the *LiDOng* cluster. The outlier trail extends up to 1179 s for individual experiments.

1179 s for individual experiments that ran into a timeout with active simulator CPU usage. The spreading of runtimes in two distinct heaps between 0 and 1.0 s is in part owed to different simulator runtimes of the eCos benchmarks, but primarily the host-CPU speed diversity in the *LiDOng* cluster.

4.6.7.4 *Post-Injection Analysis*

In the baseline assessment (Section 4.6.3), FAIL*'s symbol-level analyses (see Section 4.5.5.2) guided the GOP's initial development direction towards a configurable, generic hardening method. In the optimization and final evaluation phases, I used the global metrics described in Section 4.5.5.1.

4.7 CASE STUDY: RETURN-ADDRESS PROTECTION

The second case study on a SIHFT mechanism called *Return-Address Protection* (RAP) shares many commonalities with the GOP case study (see Section 4.6). The main commonalities and differences are the following:

- Instead of the GOP, in this case study I examined the effectiveness and efficiency of a SIHFT mechanism protecting the return address and the frame pointer that are stored on the stack on the x86 architecture. This mechanism was designed and implemented by Christoph Borchert [BSS13b].
- I used a simplified fault model of byte-aligned eight-bit burst faults in memory, which invert all eight bits at a memory address at once,

instead of the single-bit flip fault model used in Section 4.6. For information-redundancy based SIHFT mechanisms that tolerate this kind of multi-bit error, the evaluation results turn out similar to single-bit flips [BSS13a], but require eight times less FI experiments for full fault-space coverage.

- Additionally to SDCs, timeouts, and CPU exceptions, I considered memory accesses of the workload to addresses outside the areas touched in the golden run as *failures.* In a real-world embedded-system setting without memory protection, these accesses could potentially have arbitrary types of adverse effects.
- The RAP case study focuses on the eCos kernel and the same set of kernel-test programs as the GOP case study (see Section 4.6.2 respectively Table 4.2), and uses the same campaign and experiment definitions. The latter is only enhanced by the capability to inject eight-bit burst faults, and to observe irregular memory accesses.

Due to this overlapping with the GOP case study, I omit the "Fault and Failure Model" and "Workload and Experiment Setup" sections in this case study, an refer the reader to Section 4.6.1 and Section 4.6.2.

Note that the original paper on the return-address protection [BSS13b] contains much more details on the AspectC++ implementation of the RAP aspect, and efficiency measurements. As these contributions are not my own, and are not relevant for demonstrating FAIL*'s capabilities, I omit them here and refer the reader to the original paper [BSS13b].

4.7.1 *Baseline Assessment*

Similar to the GOP case study, as a starting-point assessment I ran a pre-injection analysis and FI campaigns for the unmodified *baseline* variants of the workloads listed in Table 4.2 – only this time with eight-bit burst faults. Just as in the GOP case study, the RAP developer provided ELF binaries for these baseline benchmarks.

I imported the symbol tables of each binary, and generated per-symbol aggregations (see Section 4.5.5.2). Table 4.5 exemplarily lists the top ten symbols that caused the MUTEX1 benchmark to fail with an SDC, a CPU exception, a timeout, or an irregular memory access, which are subsumed in the #FAILURES column. Just as in the single-bit results shown in Table 4.3, the global `stack` tops the failure-count ranking in Table 4.5, but even with a more distinctive distance to the second-ranking `thread_obj` data structure for the eight-bit burst-fault results.

A first explanation for the `stack` being the most critical data structure for this workload is evident: Its size of 10 224 bytes offers a large attack surface compared to the next-ranking symbols – although only 69.0 percent of the stack are used during the *golden run.* But a closer look at an excerpt of the stack area of the MUTEX1 FI results in Figure 4.15 – a diagram type I call a *fault-space plot* – reveals more details. The larger diagram covers the the

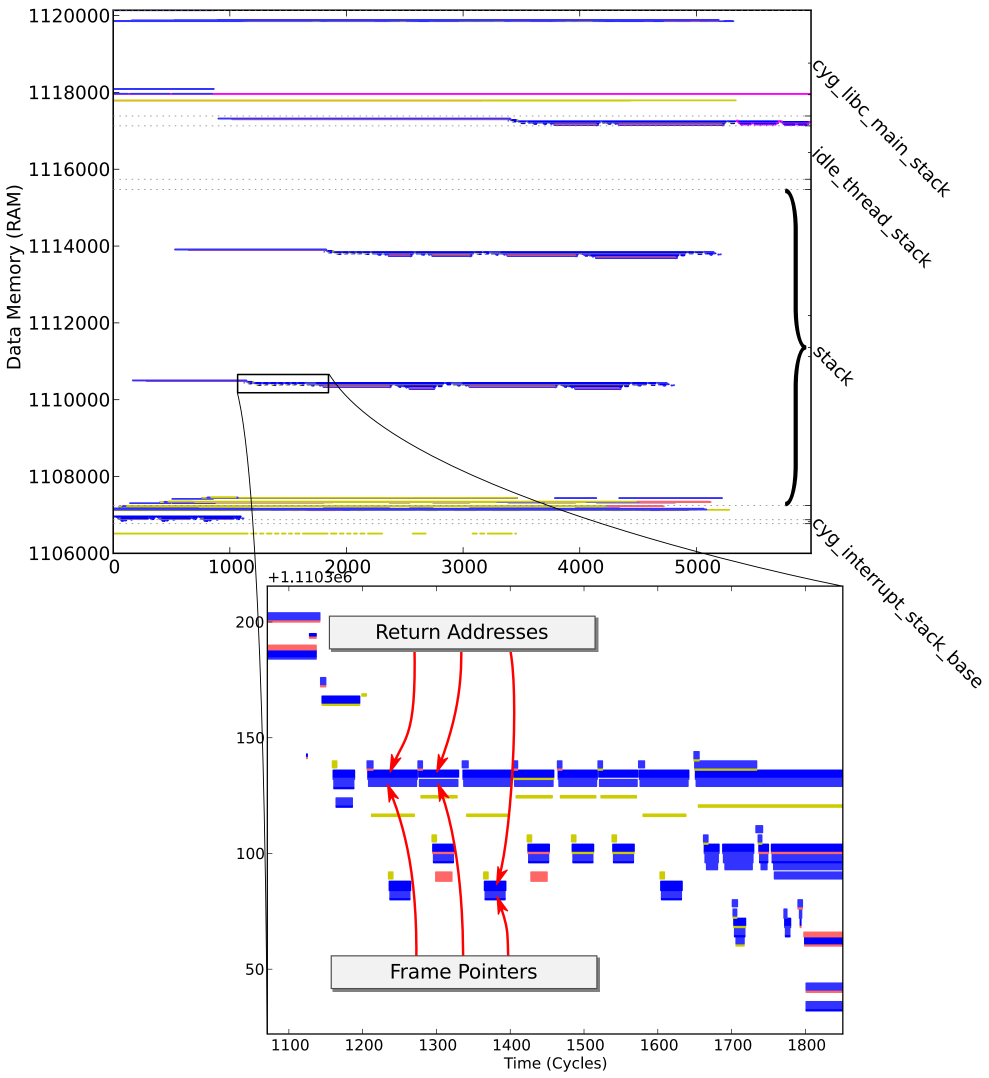


Figure 4.15: Fault-space plot showing results from the FI campaign with the unmodified MUTEX1 benchmark: Each point denotes the outcome of an independent run after injecting a burst bit-flip at a specific time and data-memory coordinate. Injections in *white* areas have no observable effect. *Blue* marks illegal memory accesses and jumps. CPU exceptions are colored *red* and timeouts *yellow* respectively. *Magenta* data points show benchmark runs that finish, but yield wrong output (SDC). *Adapted from [BSS13b].*

SYMBOL	ADDRESS	SIZE	#FAILURES	%
`stack`	`0x10E7E0`	10 224	2 377 760	42.2 %
`thread_obj`	`0x10E680`	352	984 232	17.5 %
`Cyg_Scheduler::scheduler`	`0x111800`	132	445 920	7.9 %
`cvar1`	`0x10E650`	8	221 888	3.9 %
`m1`	`0x10E63C`	12	176 448	3.1 %
`cyg_interrupt_stack_base`	`0x10E410`	512	175 424	3.1 %
`m1d`	`0x110FE0`	4	170 784	3.0 %
`Cyg_Interrupt::dsr_list`	`0x1117EC`	4	170 208	3.0 %
`m0`	`0x10E630`	12	169 248	3.0 %
`cvar2`	`0x10E658`	8	162 048	2.0 %

Table 4.5: 8-bit burst FI results for the RAP case study: Top ten fault-susceptible symbols for the unmodified MUTEX1 benchmark.

complete benchmark runtime in its horizontal dimension (5986 simulated CPU cycles), and all of data memory (20 111 bytes) in its vertical dimension. Each point in this diagram denotes the outcome of an independent FI run after injecting a burst bit-flip at the specific time and data-memory coordinate of the point. Injections in *white* areas have no observable effect, *blue* marks illegal memory accesses and jumps, CPU exceptions are colored *red*, and timeouts *yellow* respectively. *Magenta* data points show benchmark runs that finish, but yield wrong output (SDC).

The excursus on the next page provides more details on the IA-32 stack-frame layout.

The zoomed-in section reveals a primary reason the stack is so susceptible to faults: Memory corruptions in return addresses and frame pointers lead to a crash with certainty when the function returns to its caller after the FI. A more thorough manual analysis exposes that this is by far not the only, but the most homogeneous and widespread reason for crashes after faults in the stack memory.

4.7.2 *Return-Address Protection*

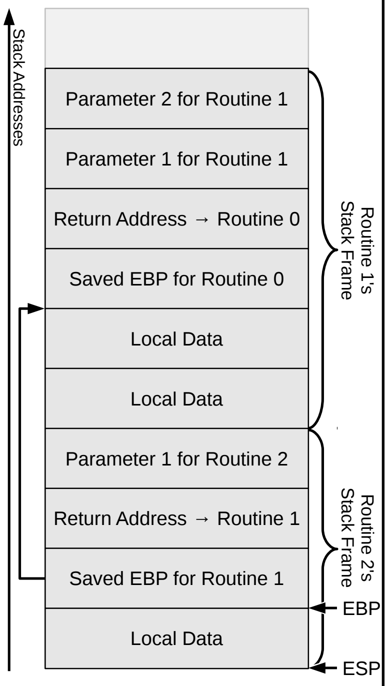
To address this problem, the RAP developer chose a compiler-based approach that stores redundant copies of return address and frame pointer directly after entering any function, and compares and restores them right before returning to the caller. However, the MUTEX1 call-stack histogram in Figure 4.16 suggests that it may be too naïve to protect *all* functions: Some are called very few times and execute for a long time (darker colors), others are called very often or run very shortly (brighter). Intuitively, the latter

EXCURSUS: IA-32 STACK FRAMES

As virtually all high-level programming languages support the nested invocation of reusable code fragments – such as functions, methods, or procedures – modern CPUs have a built-in awareness of a call stack and special instructions for its manipulation. For instance, IA-32 CPUs – as used in all case studies in this chapter – have a `call` and a `ret` instruction as well as a stack-pointer register `ESP`:

- Whenever a function is called, the `call` instruction decrements the stack pointer, saves the current instruction pointer (`EIP`) – the *return address* – on the stack, and loads the address of the target function into the instruction pointer register.
- Typically being the last instruction of a called function, the `ret` instruction reads the return address from the stack, increments the stack pointer, and loads the address into the instruction pointer.

Besides saving return addresses, in the widely used *cdecl* calling convention the stack on x86 is also used for function-call parameters and local variables. This leads to a stack-memory layout that consists of so-called stack frames – one for each active function. On x86, the register `EBP` is used to reference parameters and variables in the stack frame of the current function. `EBP` – also called *base pointer* or *frame pointer* – contains a copy of the `ESP` register created at the beginning of a function execution. To be able to restore the previous `EBP` register value directly before returning with `ret`, it is saved on the stack as well. Thereby, all stack frames are linked. Being a dynamic, pointer-linked data structure that stores code addresses, it is no surprise that memory corruptions can cause program misbehavior or even crashes.



Other CPU types than IA-32, especially RISC CPUs, have dedicated CPU registers for saving the return address, and also for passing parameters and holding local variables. However, these registers only help the leaf functions in the function call tree. Other functions need to use an in-memory stack, just as on x86. Therefore, these architectures can be expected to be less susceptible to memory errors, but failures caused by memory corruptions on the stack still exist.

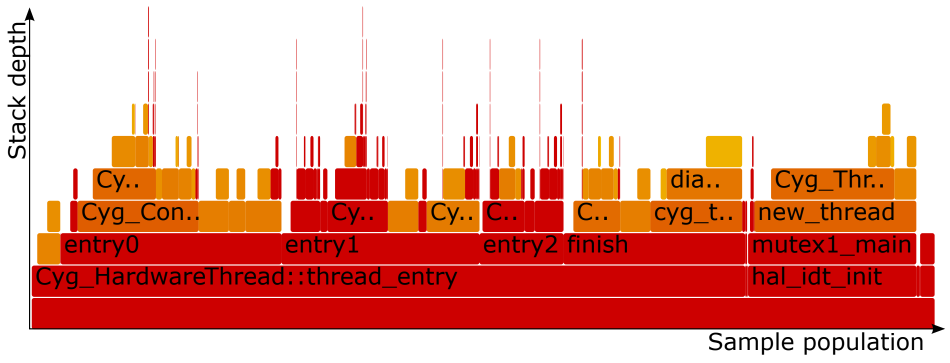


Figure 4.16: MUTEX1 benchmark call-stack histogram in form of a *flame graph*: Each bar represents a function's stack frame, and the x-axis denotes the function's runtime in the same stack context (but does *not* show the passing of time from left to right – it is actually ordered alphabetically). Lighter colors show a higher number of calls per runtime: Dark-red functions are long-running and only called once, orange to yellow ones run shorter and/or are called multiple times. *Adapted from [BSS13b].*

sort should be considered to be run without additional protective code, as it would quickly worsen the runtime and static code-size overhead – even beyond the break-even point where the additional exposure to faults outweighs all gains from being protected, as observed in the GOP case study (see Section 4.6).

For the eCos kernel-test benchmarks used in this case study, the RAP developer chose to protect those functions that had a runtime accounting for at least 0.1 percent of the runtime sum of all functions of the particular benchmark. Additionally, he added the constraint that a function must execute for at least 50 clock cycles on average to be added to the list of protected functions. Over all benchmarks, the length of this list amounts to an average of nine protected functions.

Like in the GOP case study, the developer used AspectC++ to modularize the RAP, and make it configurable with the benchmark-specific list of functions to protect. In essence, immediately after a function's entry the aspect stores redundant information on the return address and the frame pointer on the stack. Right before the function leaves, the implementation checks whether the redundancy still matches the originals, and – depending on the RAP variant – signals an error detection or repairs the error. Using this basic scheme, the RAP developer implemented two variants:

DETECTION: This variant uses a simple checksum to detect errors in both the return address and the saved frame pointer. On checksum mismatch, the running thread is aborted (fail-stop behavior).

CORRECTION: In this variant, the RAP aspect creates two copies of the return address and the saved frame pointer, yielding TMR with majority voting. Errors are transparently repaired if a majority can be established.

SYMBOL	ADDRESS	SIZE	#FAILURES	%
`stack`	`0x10EE60`	10 224	1 805 912	29.4 %
`thread_obj`	`0x10ED00`	352	1 347 976	22.0 %
`Cyg_Scheduler::scheduler`	`0x111E80`	132	629 024	10.2 %
`cvar1`	`0x10ECD0`	8	307 360	5.0 %
`m1`	`0x10ECBC`	12	243 464	4.0 %
`m1d`	`0x111660`	4	237 504	3.9 %
`Cyg_Interrupt::dsr_list`	`0x111E6C`	4	236 416	3.8 %
`m0`	`0x10ECB0`	12	232 896	3.8 %
`cvar2`	`0x10ECD8`	8	224 832	3.7 %
`cvar0`	`0x10ECC8`	8	199 328	3.2 %

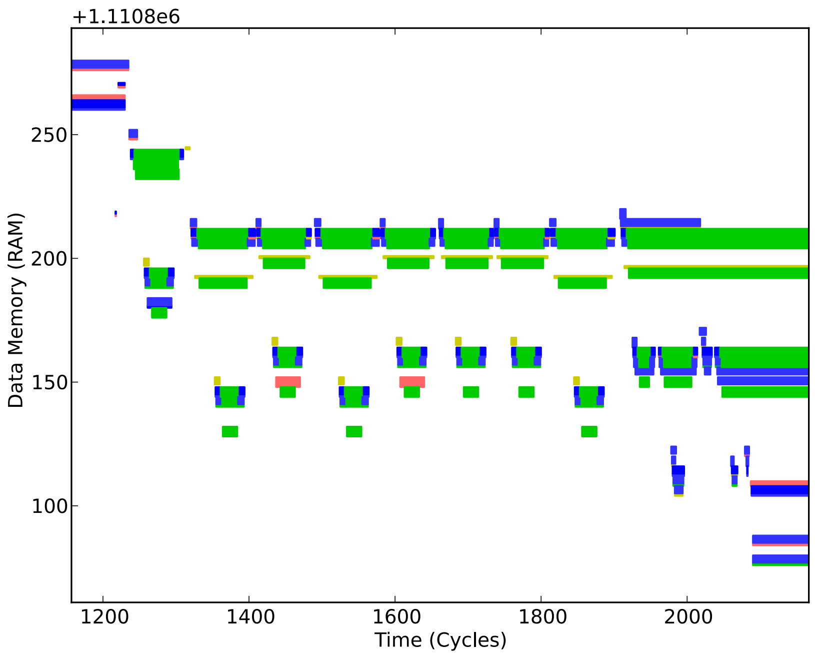
Table 4.6: MUTEX1 top ten susceptible symbols in the Correction variant: The number of crashes resulting from faults on the main stack are reduced by 24.0 % in comparison to the baseline variant (cf. Table 4.5) for this particular benchmark. Note the increase in failures for `thread_obj` by 37.0 %, resulting from longer exposure by the pathologically high runtime overhead for this tiny benchmark.

The RAP developer compiled the eCos kernel-test benchmarks in a Baseline, a Detection, and a Correction variant for IA-32 with the GNU C++ compiler (GCC, eCosCentric GNU tools 4.3.2-sw, optimization level `-O2`), and provided ELF binaries for the evaluation. eCos 3.0 was configured just as in the GOP case study (see Section 4.6.2).

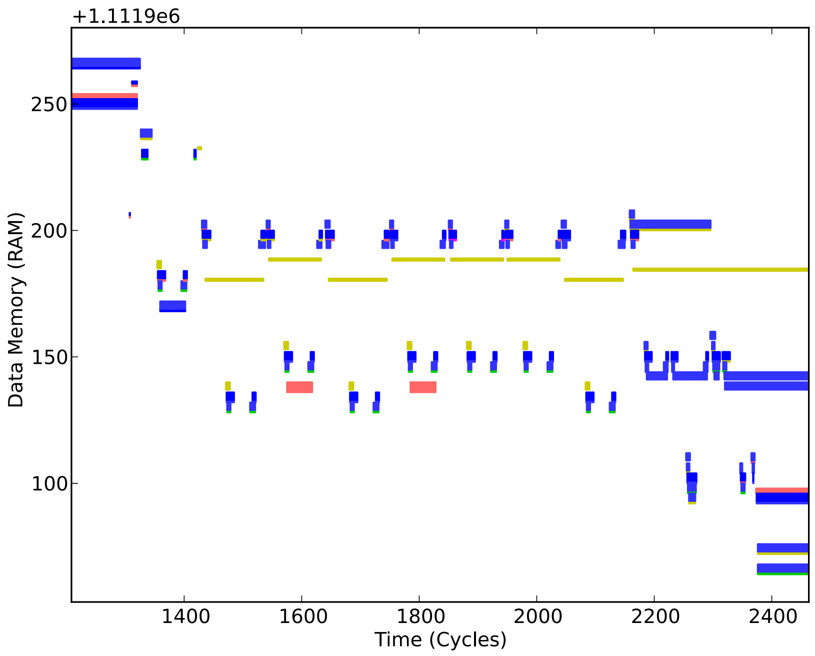
4.7.3 *Evaluation: Detection and Correction Effectiveness*

The FI campaign covering the full memory-fault space for all 21 benchmarks – each in its Baseline, Detection, and Correction variants – consisted of about 3.1 million single experiment runs and amounted to 1673 simulation hours. Running more than 100 FAIL* clients in parallel on the faculty's student-pool machines reduced the wall-clock campaign runtime to below one day.

The fault-space plots in Figure 4.17 give a qualitative impression of the FI campaign results: The Detection variant (Figure 4.17a) detects almost all faults in return addresses and frame pointers that led to crashes before (cf. Figure 4.15), and then fail-stops. The Correction variant (Figure 4.17b) additionally transparently corrects these faults. Only very short susceptible time frames between function call and information-redundancy creation, and between check/repair and function return remain – a phenomenon known as *Time-of-Check-to-Time-of-Use* (TOCTTOU) in the context of system security



(a) Detection variant: *Green* results denote experiments where the fault was successfully detected (fail-stop). In the baseline variant (cf. Figure 4.15), all of these led to crashes.



(b) Correction variant: Almost all return address and frame-pointer related crashes are masked; only very short susceptible time frames between call and replica creation, and between check/repair and function return remain (TOCTTOU phenomenon).

Figure 4.17: MUTEX1 benchmark, close-up of the same stack area as shown in Figure 4.15, for both Detection and Correction variants. *Adapted from [BSS13b].*

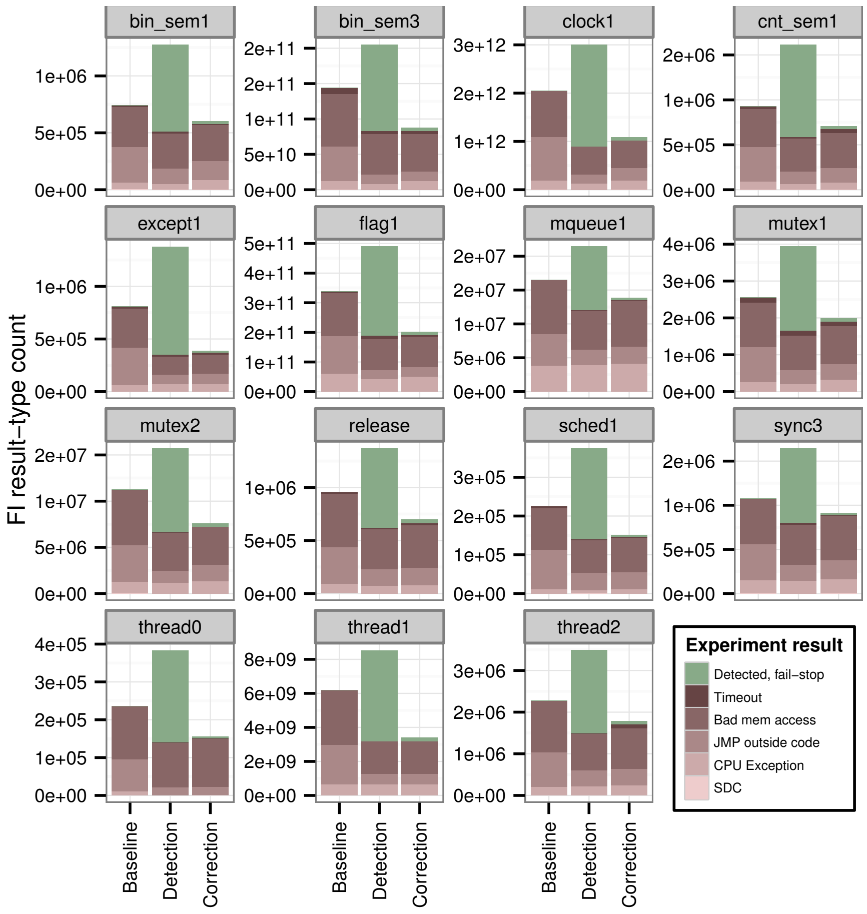


Figure 4.18: Stack-only, absolute failure type (and detection) counts from the RAP FI campaign (BIN_SEM2, CLOCKCNV, CLOCKTRUTH, KILL, MUTEX3 and SYNC2 benchmarks omitted due to their extremely long runtime): Both EDM/ERM variants improve resiliency to faults on the stack(s) substantially.

and race conditions [McP74]. The top ten susceptible-symbols list for the MUTEX1 benchmark in the Correction variant (Table 4.6) still ranks the main application `stack` first, but with a 24.0 percent reduction in the failure count.

The FI-campaign results for the remaining benchmarks, filtered to only include injections in the different stacks, and differentiated into the observed experiment outcomes, are shown in Figure 4.18. On average, the Detection variant reduces stack failures by 53.8 percent: Especially the long-running benchmarks profit from the EDM. Notably, the green “detected” bars exceed the baseline in all cases – by 46.4 percent on average. Besides an increased attack surface due to the additionally executed instructions, I assume false DUEs (see Section 2.3.2.4) to be the primary reason for this phenomenon: The caller function’s saved frame pointer may not be used after it is restored. In this case, an error in the saved frame pointer has no effect in the Baseline variant, but triggers an error detection in the Detection variant.

The excursus on page 131 provides more details on the IA-32 stack-frame layout.

The Correction variant is similarly effective: On the average, it masks 45.5 percent of all stack failures, and detects another 3.2 percent. The in-

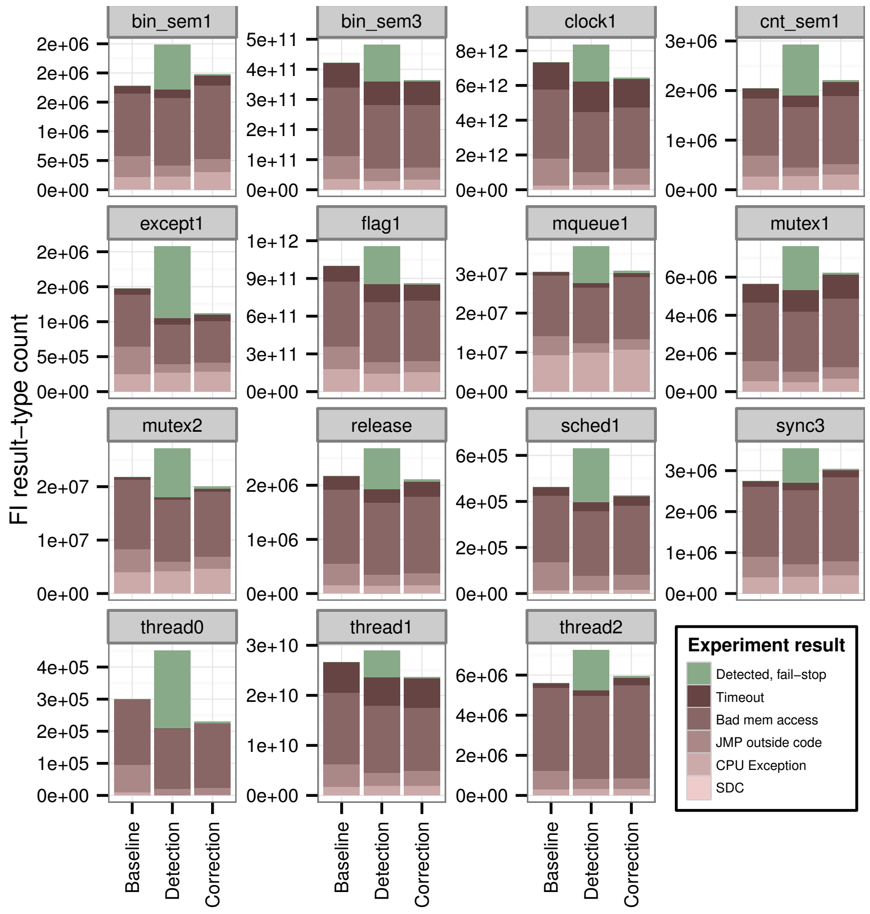


Figure 4.19: RAP FI results as in Figure 4.18, but covering the complete fault space including data structures outside the stack: Without additional protection, the life-time increase for data structures other than return addresses and frame pointers exceeds all gains from the RAP in some cases.

creased attack surface pays off even for the shortest-running benchmarks, and the aforementioned false DUEs become rather improbable.

However, the FI results from the *complete* fault space in Figure 4.19 reveal that the *total* failure-count reduction is not as significant as for the stack-only results (Figure 4.18). In case of BIN_SEM1, CNT_SEM1, MUTEX1, SYNC3, and THREAD2, the Correction variant even shows slightly *more* failures than the Baseline. The reason is that the additionally executed TMR code increases the life time – and, hence, the attack surface – of data structures outside the stack. For the aforementioned five benchmarks, this lifetime increase reaches an extent that failures originating from these data structures exceed the gains from reducing the failure count within the stack. For example, the failure count relating to the `thread_obj` data structure grows by 37.0 percent from the Baseline (Table 4.5) to the Correction variant (Table 4.6).

Nonetheless, this does not mean that the RAP is generally not applicable to these benchmarks, but that a separate protection mechanism – such as the GOP (see Section 4.6) – is needed for data structures outside the stack to achieve an overall improvement.

4.7.4 *Conclusions on Fail* Tooling*

From the perspective of the Fail* evaluation infrastructure – besides the analyses already demonstrated in the GOP case study, – the primary fine-grained analysis method I additionally applied was the fault-space plot feature, as shown in Figure 4.15 and Figure 4.17. Fault-space plots allow to visually inspect large amounts of FI results, and to gain insights about particularly vulnerable data structures and program phases, or phenomenons like TOCTTOU. Additionally, filtering the FI results for addresses within the stack areas of the target workloads allowed to contrast the failure-count reductions of the RAP aspect on the stack (Figure 4.18), and the failure-count increases in the remaining memory areas due to the additional runtime (Figure 4.19).

4.8 CASE STUDY: WSOS / BARE-METAL-SORT

The third case study conducted with the Fail* FI infrastructure is from a purely educational context. In February 2016, I contributed a practical "hands-on" session on "Fault-Injection based Assessment of Software-Implemented Hardware Fault Tolerance" to the *Winter School on Operating Systems* (WSOS) in Graz/Austria.[16] The goal of this hands-on session was to introduce the postgraduate-student participants to FI by iteratively improving a given bare-metal sorting program, embedded in a small competition for the lowest SDC and total-failure counts.

The following sections describe the setup and teaching material we used in the hands-on session, present the coarse- and fine-grained baseline-assessment techniques used by the participants, and discuss conclusions on using Fail* in an educational context.

4.8.1 *Hands-On Setup*

After some introductory words on FI in general and Fail* in particular, we provided the participants with VirtualBox VM images and handouts with basic instructions. The Debian-based VM image contained all ingredients needed for developing a small, fault-tolerant bare-metal program, injecting faults into it with Fail*, and analyzing the results on different levels of granularity:

- The *bare-metal sort* program, including sources and small wrapper scripts for building, tracing, importing, and FI.

16 My colleagues Michael Lenz and Christian Dietrich helped creating supplemental materials for the hands-on tutorial.

- A complete FAIL* repository clone, with pre-built, FAILBochs-based *Generic-Tracing* and *Generic-Experiment* binaries (see Section 4.5.2) and ready-to-use post-injection analysis scripts.
- A MariaDB installation for FI job and result data.

The given *bare-metal sort* workload was a simple, bare-metal IA-32 implementation of a sorting program, taking a statically linked array as an input, sorting the data in place with the bubble-sort algorithm, and writing the sorted data to an I/O port observable by the *Generic-Tracing* experiment. The participants were instructed to conduct an FI-based baseline assessment using different coarse- and fine-grained analysis methods, and to then iteratively improve *bare-metal sort*.

4.8.2 *Fault and Failure Model*

As the participants were supposed to run complete FI on their resource-limited laptops, we decided to use a similar fault model as in the RAP case study – uniformly-distributed 8-bit burst faults in main memory. Compared to single-bit flip faults, the factor of eight lower number of FI experiments helped keeping the development-iteration frequency high enough during the hands-on session.

The chosen failure model included SDCs, timeouts, CPU exceptions, and as an additional category "detected" faults. We set two separate goals for the session's fault-tolerance competition: minimization of the SDC count, and minimization of the sum of all failure categories, including "detected". While the former goal aims at *safety*, the latter additionally incorporates *reliability* (see Section 2.2.1).

4.8.3 *Baseline Assessment and Analysis Methods*

By directly using pre-built binaries for *Generic-Tracing* and the *Generic-Experiment*, the hands-on participants were spared defining an own experiment procedure, and could start with a baseline assessment right away. They compiled the *bare-metal-sort* baseline with Debian GCC 4.9.2-10, ran the usual pre-injection steps – golden run with Generic-Tracing, trace import, and def/use pruning, – and conducted the FI campaign with 2879 experiments on their laptops in a time frame of a few minutes.

After the FI campaign, the participants could freely choose from FAIL*'s palette of post-injection analysis methods, including:

- The *global aggregation* for the different experiment-outcome types (see Section 4.5.5.1). Table 4.7 shows the result-count numbers for the *bare-metal-sort* baseline – these numbers were the basis for the competition, and meant to be beaten by the participants' modifications to the program. The participants were also given the possibility to calculate the fault-coverage metric, but advised not to use the resulting numbers as a basis for workload-variant comparison.

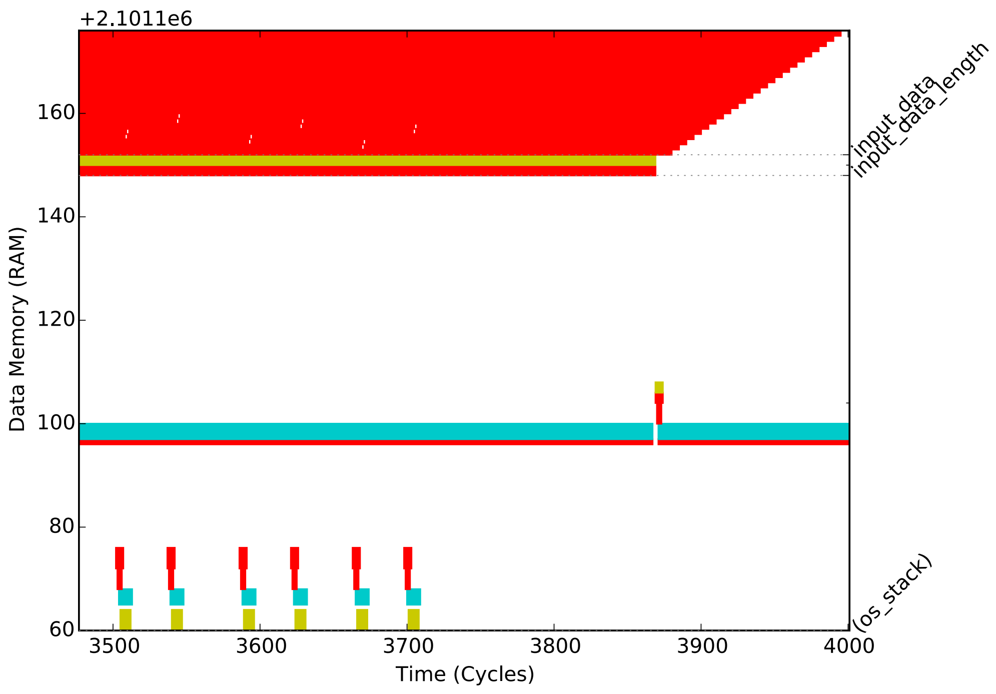
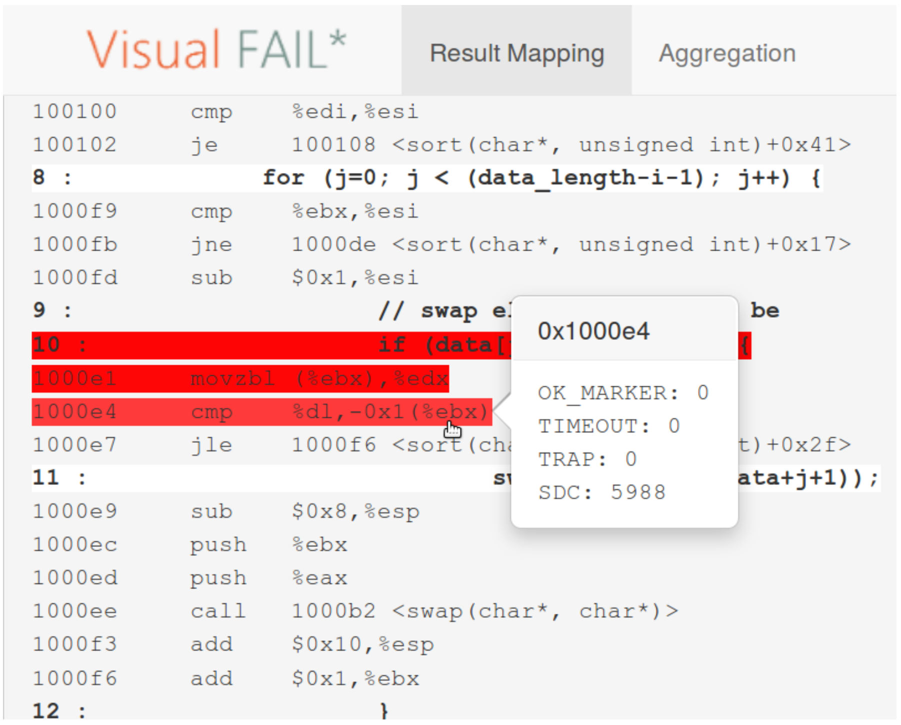


Figure 4.20: Fault-space plot excerpt for the *bare-metal-sort* baseline: SDC failures in the input_data array stand out most prominently. The slope towards the benchmark's end stems from the output loop, after which each array member is not used anymore. *Red* data points show SDC results, *yellow* means timeout, *turquoise* marks CPU exceptions.



```
100100      cmp      %edi,%esi
100102      je       100108 <sort(char*, unsigned int)+0x41>
8  :                for (j=0; j < (data_length-i-1); j++) {
1000f9      cmp      %ebx,%esi
1000fb      jne      1000de <sort(char*, unsigned int)+0x17>
1000fd      sub      $0x1,%esi
9  :                         // swap e                    be
10  :                        if (data[                    {
1000e1      movzbl   (%ebx),%edx
1000e4      cmp      %dl,-0x1(%ebx)
1000e7      jle      1000f6 <sort(cha                     t)+0x2f>
11  :                                  s                  ata+j+1));
1000e9      sub      $0x8,%esp
1000ec      push     %ebx
1000ed      push     %eax
1000ee      call     1000b2 <swap(char*, char*)>
1000f3      add      $0x10,%esp
1000f6      add      $0x1,%ebx
12  :                        }
```

```
0x1000e4

OK_MARKER: 0
TIMEOUT: 0
TRAP: 0
SDC: 5988
```

Figure 4.21: VisualFAIL* highlights *bare-metal sort* source-code lines and associated assembler instructions: The highlighted code indicates that the bubble-sort implementation's inner loop activates many faults that lead to SDCs.

OUTCOME	RESULT COUNT
No Effect	215 973
Timeout	11 277
CPU Exception	15 336
SDC	109 766

Table 4.7: Global result counts for the *bare-metal-sort* baseline.

OUTCOME	SYMBOL	SIZE	RESULT COUNT
No Effect	`os_stack`	4096	213 337
	`input_data`	24	2096
	`input_data_length`	4	540
Timeout	`input_data_length`	4	7746
	`os_stack`	4096	3531
CPU Exception	`os_stack`	4096	15 336
SDC	`input_data`	24	94 000
	`os_stack`	4096	8036
	`input_data_length`	4	7730

Table 4.8: Per-symbol result counts for the *bare-metal-sort* baseline.

- *Per-symbol aggregations* for the different experiment-outcome types (see Section 4.5.5.2). The baseline results shown in Table 4.8 quickly pointed out to the participants that faults in the `input_data` array are those primarily responsible for SDCs, and CPU exceptions originate only from the `os_stack`.

- *Fault-space plot* with symbol annotations. The fault-space excerpt in Figure 4.20 confirmed the criticality of the `input_data` and `os_stack` variables/data structures, and also indicated the partitioning of the workload into two phases: sorting (until approximately cycle #3870), and output of the `input_data` array. Faults in the array elements have no effect after they have been printed to the output channel, explaining the slope in the fault-space plot throughout the output loop.

- *Per translation-unit aggregations* (see Section 4.5.5.3). Matching the results from the per-symbol aggregation and the fault-space plot, the per translation-unit aggregation in Table 4.9 directed the participants' attention towards the bubble-sort implementation in `bubblesort.cc`, and the output loop in `output.cc`, which activate most of the faults that lead to SDCs.

OUTCOME	TRANSLATION UNIT	RESULT COUNT
No Effect	`bubblesort.cc`	82 192
	`main.cc`	46 976
	`fail.cc`	11 332
	`sort.cc`	1769
	`output.cc`	600
Timeout	`main.cc`	7746
	`sort.cc`	3521
	`output.cc`	10
CPU Exception	`bubblesort.cc`	11 571
	`sort.cc`	3375
	`output.cc`	390
SDC	`bubblesort.cc`	69 215
	`output.cc`	26 294
	`main.cc`	7730
	`fail.cc`	4000
	`sort.cc`	2527

Table 4.9: Per translation-unit result counts for the *bare-metal-sort* baseline.

OUTCOME	FUNCTION	RESULT COUNT
No Effect	`sort(char*, unsigned int)`	82 336
	`main`	47 040
	`swap(char*, char*)`	12 957
	`print_data(char*, unsigned int)`	536
Timeout	`main`	7746
	`swap(char*, char*)`	3500
	`sort(char*, unsigned int)`	21
	`print_data(char*, unsigned int)`	10
CPU Exception	`sort(char*, unsigned int)`	11 571
	`swap(char*, char*)`	3375
	`print_data(char*, unsigned int)`	390
SDC	`sort(char*, unsigned int)`	69 242
	`print_data(char*, unsigned int)`	26 294
	`main`	7730
	`swap(char*, char*)`	6500

Table 4.10: Per-function result counts for the *bare-metal-sort* baseline.

TEAM	BARE-METAL-SORT CHANGES	SDC COUNT
Baseline	–	109 766
"Kuan"	inline `swap()`	65 089
"AT"	inline `swap()`, selection-sort	60 556
"Kuan"	shellsort, optimization `-O3`	4462
"Jana"	shellsort, inl. `swap()`, checksum after output	2276
"Jana"	+ checksum (four bytes at once)	0

TEAM	BARE-METAL-SORT CHANGES	NON-OK COUNT
Baseline	–	136 379
"Kuan"	inline `swap()`	77 314
"AT"	inline `swap()`, selection-sort	75 496
"Kuan"	shellsort, optimization `-O3`	61 271
"Björn"	inline `swap()`, optimization `-O3`	49 294
"Björn"	+ MSB check / inversion	32 457
"AT"	inl. `swap()`, selection-sort, TMR	28 335

Table 4.11: The two WSOS FI hands-on competition leaderboards: One for the SDC-minimization goal, and one for the goal of minimizing all non-"No Effect" results.

- *Per-function aggregations.* As to be expected from the per translation-unit aggregation, `sort()` and `print_data()` activate the most SDC-prone faults (Table 4.10).
- *Instruction-level and source-code level aggregations* in the web-browser based VisualFAIL* tool. The result detail in Figure 4.21 indicated – with little surprise for a sorting benchmark – that the machine instructions for element comparison, respectively the high-level C++ language construct `if (data[j] > data[j+1])`, activate many SDC-causing faults.

Based on these analysis results, the hands-on participants began to iteratively improve bare-metal-sort.

4.8.4 *Competition Results*

Although each participating team worked on their own *bare-metal-sort* variant, due to the scenario's simplicity – and the initial, open discussion on possible SIHFT solutions – several teams chose similar approaches in different variants and combinations.

- Some teams switched to more efficient sorting algorithms, such as shellsort or selection-sort. The reduced execution time directly reduces the time component of the "attack surface", and thus, the SDC count.
- Another widely-used technique was to inline the `swap()` function. As this function is called very frequently by the sorting algorithm to exchange two array elements, this, again, significantly reduces execution time and SDCs, but also avoids repeatedly storing a return address on the `os_stack`. The removal of this instruction-address indirection measurably reduces the CPU-exception count.
- Choosing a higher optimization level (GCC's `-O3` switch) has a similar effect: It reduces execution time, and potentially eliminates some of the state-based indirections.
- Some teams approached the problem with classic redundancy techniques, such as TMR and voting, or checksumming.

Besides these generally applicable techniques, some teams also discovered loopholes in the scenario setup and the competition rules. For example, one team figured out that eight-bit burst faults can easily be detected and reverted if the `input_data` array is known to contain only values up to 127. Another group exploited the fact that the workload could signal "detected" even after all data was already printed to the output channel, allowing to even reduce the SDC count to zero.

Table 4.11 shows the two final WSOS hands-on leaderboards as they developed throughout three hours of the competition.

4.8.5 *Conclusions on Fail* Tooling*

The Fail* tooling – especially the assessment-cycle layer amenities, like the ready-to-use *Generic-Tracing* and *Generic-Experiment* tools, and the various sophisticated analysis scripts – proved to be well-suited for the context of a practical hands-on lecture:

- Not least due to Fail*'s pruning techniques, the tooling is sufficiently efficient for running even complete fault-space scans for a small example workload on contemporary laptop-class computers.
- The assessment-cycle layer tools and abstractions allow for a learning curve adequate for a time-constrained lecture with non-experts.

- FAIL*'s coarse- and fine-grained analysis methods provide all necessary input for purposeful and targeted SIHFT hardening. Although the bare-metal-sort workload would certainly have been manageable without the more sophisticated analyses, this approach would have been slower and would have involved guesswork.

On a didactic side note, the competition character of the hands-on session proved extremely motivating to the participants.

4.9 DISCUSSION

The design and implementation of FAIL*, as presented in this chapter, achieves most of the goals I set in Section 4.1.

4.9.1 *Fault-Space Coverage*

FAIL* is capable of covering the complete ISA-level fault-space of a given workload. This goal was achieved by integrating parallelization as a core feature into FAIL*'s plumbing layer (see Section 4.3), and by adding fault-space pruning as an integral step to the assessment-cycle layer tooling (see Section 4.5).

See Chapter 5 for details on the Fault-Similarity Pruning (FSP) heuristic, which further reduces the FI efforts for complete fault-space coverage.

As a result, FAIL*'s post-injection analyses (see Section 4.5.5) can provide fine-grained analyses down to the level of program modules, functions, variables, source-code lines, or machine instructions. The case studies in Sections 4.6 to 4.8 demonstrated the utility of these analyses for iteratively developing, improving, and placing/configuring SIHFT mechanisms.

4.9.2 *Measurement and Comparison*

See Chapter 7 for details on the extrapolated absolute failure count *metric and its derivation.*

The case studies also demonstrated that FAIL* can be used for comparing different workload variants – and deciding whether a SIHFT-protected program reduces the failure probability compared to the unhardened program, – as it implements the *extrapolated absolute failure count* metric.

4.9.3 *Maintainable Back-End Extension*

The goal of extending existing simulator back-ends in a maintainable way using AOP was achieved *partially*.

As described in Section 4.3.2.3, the control-flow transfers from the Bochs x86 simulator to FAIL* were implemented using AspectC++, successfully achieving separation of simulation and FI concerns, and aspect quantification – especially for capturing the various different types of memory accesses within Bochs. Thereby, the scattering and tangling effects mentioned in Section 3.5.3 could be avoided. However, in one instance Bochs had to be extended by an explicit "hook function" call, as the exact location within the

FAULT MODEL	PREPARATION	EXPERIMENT IMPLEMENTATION	PUBLICATIONS
Transient *single*-bit flips in *data* memory, uniform	Golden-run memory-access trace analysis (using *Tracing* plugin and `import-trace` tool)	*Fault trigger*: *N* CPU cycles passed *Fault injection:* Memory or CPU abstraction in EEA layer	e.g., [BSS13a, HBD^{+}14, BSS15, HLDL15, SBS15, DHL15, BS15]
Transient *single*-bit flips in *data* memory, on access *("inject-on-read")*			[SBS15]
Transient *multi*-bit flips in *data* memory, uniform			e.g., [BSS13a, SBS14]
Transient *single*-bit flips in *instruction* op-codes	Golden-run instruction trace analysis (*Tracing* plugin & `import-trace` tool)		[HUD^{+}14a, HUD^{+}14b]
Transient *single-bit* flips in *register file*, uniform	Golden-run instruction trace analysis & disassembly (*Tracing* plugin & `import-trace` tool)		[HLDL15, DHL15]
Transient *single*-bit flips in *register file*, on access *("inject-on-read")*			[HUD^{+}14a, HUD^{+}14b]
Transient *multi*-bit flips in *register file*, on access *("inject-on-read")*			
Permanent single-bit flips in memory	—	*Fault trigger:* Memory write at faulty location *Fault injection:* Memory abstraction in EEA layer (re-write faulty bit)	[SKSE13]

Table 4.12: Fault-model examples implemented using FAIL*, necessary campaign preparation steps, and EEA-layer primitives used in the respective experiment implementation.

main CPU loop where the control flow had to be diverted was not directly capturable by AspectC++'s pointcut language.

Nevertheless, I was not able to completely address one "specialist"-specific issue mentioned in Section 3.5.3: keeping in sync with the simulator's mainline evolution. It seems that the Bochs developers started an extensive overhaul of the simulator's core components soon after the 2.4.6 version that is currently part of the Fail* tree, for example the central main CPU loop was changed for performance reasons in a way that the current extension aspects do not work as expected anymore. Moving to a newer Bochs version is certainly possible, but will require more efforts than just importing a newer version of its source-code tree.

As described in Section 4.3.2.4, gem5 and QEMU could not be extended using AspectC++, as the compiler currently cannot weave into C code (QEMU) or C++ templates (heavily used in gem5). Nevertheless, the simulator source-code modifications were kept to a minimum.

4.9.4 *Back-End and FI-Technique Flexibility*

The *Execution-Environment Abstraction* (EEA) – as part of Fail*'s plumbing layer (see Section 4.3) – allows Fail* users to flexibly switch between different back-ends and FI techniques, including simulation-based FI in three different simulators, and a TAP-based/SWIFI hybrid FI method targeting an ARM development board (see Section 4.3.3).

Beyond this flexibility regarding back-end and FI-technique choice, Fail*'s plumbing layer also allows to implement a wide range of ISA-level fault models, including both CPU and memory fault models. Table 4.12 lists examples for fault models that have already been implemented using Fail*, the necessary preparation steps before running the campaign (see Section 4.5.3), the EEA respectively plumbing-layer primitives used in the experiment implementation, and the publications the results were published in.

4.10 SUMMARY

To summarize, this chapter described the design, implementation, and case-study based evaluation of Fail*, a flexible and versatile architecture-level FI tool and framework for developers designing and deploying SIHFT measures. Fail*'s basic *plumbing layer* provides a client/server infrastructure for parallel experiments, and abstracts from a concrete execution environment, such as a hardware simulator, or actual development hardware connected via a TAP. Building on the APIs provided by the plumbing layer, the *assessment-cycle layer* restricts the degrees of freedom to its users, trading them for simpler use and tool support for the complete fault-tolerance assessment cycle.

Fail* achieves complete fault-space coverage through parallelization and pruning techniques, and enables a multitude of novel fine-grained analysis techniques, as demonstrated by several case studies. The provided absolute failure count metric allows to compare different workload variants. At least in the case of FAIL*'s Bochs back-end, the FI-implementation maintainability improves significantly over the state of the art. Additionally, FAIL*'s plumbing-layer abstractions allow a high degree of flexibility regarding target back-ends, FI techniques, and ISA-level fault models.

5 FAULT-SIMILARITY PRUNING: CAMPAIGN SPEEDUP BY EXPERIMENT-COUNT REDUCTION

"Insanity is doing the same thing over and over again, but expecting different results."
— fictional character Jane Fulton in Rita Mae Brown, *Sudden Death* [Bro83]

Contents

As the previous chapter showed, some of Fail*'s novel result-analysis techniques – such as fault-space plots, symbol-level analyses, or mapping of FI results to high-level language constructs down to the level of single source-code lines – require complete fault-space coverage instead of sampling results. Unfortunately, FI campaigns exhaustively covering all possible ISA-level fault locations can cost enormous computing power – sometimes measured in quantities of CPU *years*, such as in the GOP case study (see Section 4.6), – and become infeasible for larger workloads.

As a remedy to this problem, this chapter describes a novel heuristical fault-space pruning method that massively reduces the FI-campaign runtime *and* keeps information on all possible fault locations in the workload. The following section points to required background on FI and fault-space pruning principles in previous chapters. Sections 5.2 and 5.3 describe the design and implementation of the fault-space pruning heuristic, and provide details on the necessary modifications and additions to FAIL*. Section 5.4 presents and discusses evaluation results with the eCos benchmarks already introduced in the GOP case study (see Section 4.6.2) and the *automotive* category of the MiBench [GRE⁺01] benchmark suite. Section 5.5 concludes and summarizes this chapter.

Parts of this chapter were originally published on SAFECOMP 2014 [SBS14].

5.1 THE NEED FOR FULL FAULT-SPACE COVERAGE

As already described in Section 1.2.3, there exist two fundamental approaches to reduce an FI campaign's total runtime: One aims at reducing the runtime of individual experiment runs, the other at reducing the total count of experiments to conduct. Most studies leveraging FI for test or measurement purposes resort to *statistically sampling* fault locations [LCMV09, RKK⁺08]: A randomized selection of fault locations – usually in the thousands – is used to drive the FI campaign, until statistics predict a "good enough" probability for the overall result distribution to lie within the desired confidence interval (see Section 3.1.2.5). The result is an estimate on the aggregated campaign outcome, for example a probability breakup that an experiment finishes with the expected output ("No Effect"), with an SDC, a CPU exception, or a timeout.

5.1.1 *Fine-Grained Analyses*

While an estimate on the aggregated results suffices for measuring the resiliency improvement between a benchmark's baseline and a SIHFT-protected variant, it gives no authoritative insights on critical spots and local phenomena, such as the vulnerability of specific data structures or program phases.

In contrast, the fine-grained analyses presented and demonstrated in Chapter 4 – fault-space plots, symbol-level analyses, or mapping of FI results to high-level language constructs down to the level of single source-code lines – are based on the injection of faults into *all* possible fault locations of the particular workload. Depending on the workload size, these FI campaigns can take several CPU years of simulation time with FAIL*, although the accompanying `prune-trace` tool implements the highly effective def/use pruning technique (see Sections 3.3.1.1 and 4.5.3.4).

5.1.2 *Chapter Outline*

To fill this gap, in this chapter I describe a fault-space pruning heuristic that – based on def/use pruning – reduces the FI-experiment count further *and* keeps information on all possible fault locations in the program. The basic idea stems from a simple insight: If in two FI experiments *the machine state is similar* (or *identical*) at the point in time when the fault is injected, *the ex-*

periment result will be similar (or *identical*), too. Alluding to this underlying idea, I named the heuristic *Fault-Similarity Pruning* (FSP).

The following sections describe the FSP method in detail. Its most prominent properties are:

- FSP prunes the fault space *application-specifically*, and *preserves local features* of the fault space as needed by FAIL*'s fine-grained post-injection analyses.
- FSP allows the developer to *freely trade accuracy for experimentation runtime* without suffering the primary drawback of randomized sampling – its lack of result details.

As in the RAP case study (see Section 4.7), the remainder of this chapter focuses on uniformly distributed, byte-aligned eight-bit burst faults in memory, which invert all eight bits at a memory address at once. This fault model can be seen as a simplification of multi-bit flips induced by single-event upsets (see Section 2.3.2.4), which are commonly caused by particle strikes. Despite the simple fault model I nevertheless believe that the FSP technique can be generalized to other fault models, such as single-bit flips or transient faults in CPU registers or caches.

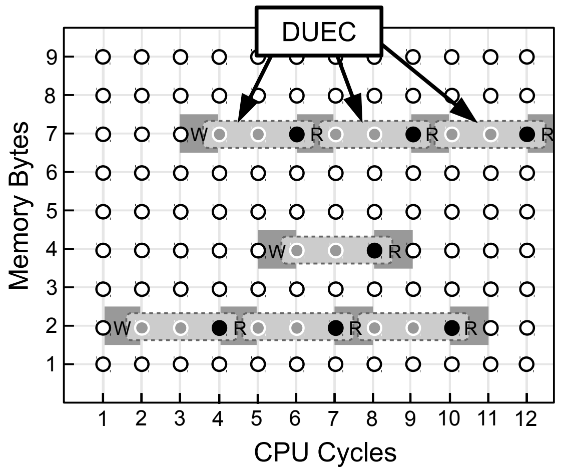
5.2 GENERALIZING FAULT EQUIVALENCE

The following section first briefly recapitulates def/use pruning and the fault-space pruning heuristics presented in Hari et al.'s Relyzer [HANR11, HANR12, HANR13] and Li and Tan's SmartInjector [LT13], which were already discussed in Sections 3.3.1.1 respectively 3.3.1.2. Then, I outline a generalization – and simplification – of the *fault equivalence* notion coined in these works. Subsequently, I derive a generic heuristic that ameliorates the inflexibilities regarding accuracy and experiment-count trade-off, the need for complex control and data-flow analyses, platform specifics, and a fixation on a single failure model (see Section 3.3.1.2).

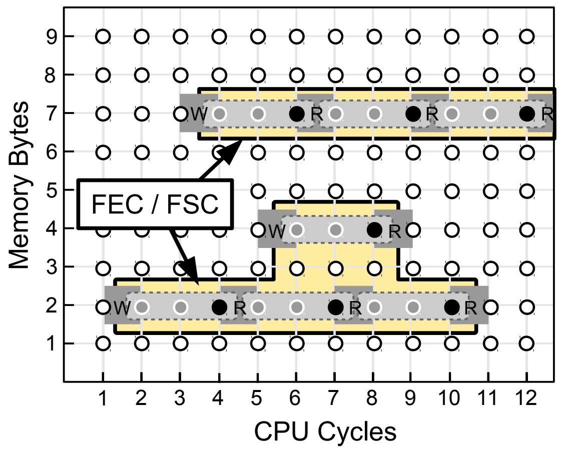
5.2.1 *Def/Use Pruning and Fault Equivalence*

As described in Section 3.3.1.1, def/use pruning exploits knowledge from a memory-access trace that was recorded during the golden run, and conservatively reduces the number of FI experiments for a complete fault-space scan without losing result accuracy. Figure 5.1a illustrates this procedure for the eight-bit burst fault model used throughout this chapter: Fault-space coordinates between a *def* or *use* (write or read), and a subsequent *use* (read) can be considered equivalent – the outcome for each injection is a priori known to be identical. Consequently, only one FI experiment is conducted for each *Def/Use Equivalence Class* (DUEC).

Building on this conservative approach, Hari et al. [HANR11, HANR12, HANR13] describe the *Relyzer* tool, providing several heuristics that *combine*



(a) Eight-bit burst bit-flip fault space and *def/use equivalence classes* (DUECs) extracted from a program trace, reducing the FI experiments (dots) that need to be conducted. Fault injections at white coordinates (non-filled circles) can be omitted, as a fault there is overwritten or never read. A black dot represents a class of equivalent faults (light-gray coordinates) between a write/read and subsequent read instruction.



(b) *Fault-equivalence classes* (FECs), or – for this chapter – *fault-similarity classes* (FSCs) group several DUECs the heuristic assumes to have the same outcome.

Figure 5.1: Def/use and fault-equivalence/fault-similarity pruning: A *Def/Use Equivalence Class* (DUEC) groups fault-space coordinates with a known identical outcome. Based on DUECs, a fault-equivalence class (FEC, Hari et al. [HANR11, HANR12, HANR13]) respectively fault-similarity class (FSC, my approach) groups DUECs it heuristically assumes to have the same outcome.

multiple DUECs into larger groups called *fault-equivalence classes* (FECs), as illustrated in Figure 5.1b. The goal of the grouping heuristics is to combine DUECs that have the same experiment outcome, as anticipated through static or dynamic analyses. In the FI campaign, from each FEC only *one* representing DUEC – the *pilot* – gets picked, and the experiment result is assumed to be identical for the remaining group members. Li and Tan [LT13] describe a similar pruning heuristic in their *SmartInjector* tool. Both approaches share a set of deficiencies:

- The approaches are inflexible regarding the result-accuracy and experiment-count trade-off: Hari et al. report an average accuracy of 96 percent, Li and Tan 94 percent. The user cannot increase the accuracy by running more FI experiments (using FECs with *less* DUEC members), or save FI-experiment efforts by accepting a lower accuracy (using FECs with *more* DUEC members).

- The grouping heuristics are based on complicated control and dataflow analyses, SPARC platform specifics, and assumptions on the experiment result interpretation: Both *Relyzer* and *SmartInjector* only differentiate between *benign* and SDC outcomes, while in many use cases – for example, the GOP case study (see Section 4.6) – more outcome types are necessary.

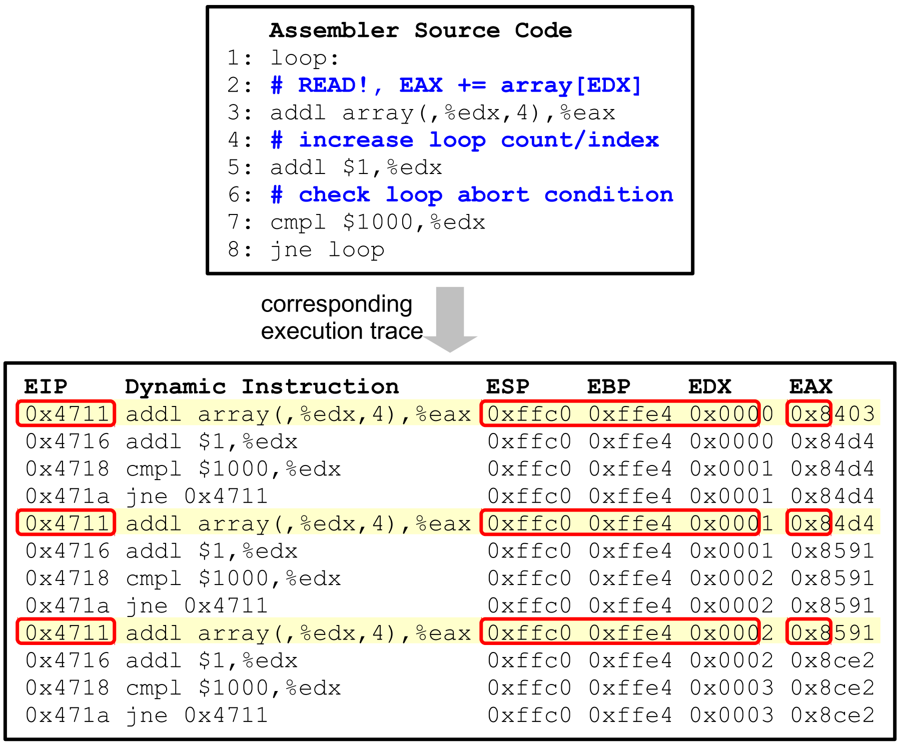
5.2.2 *Fault Similarity and a Generalization of Fault Equivalence*

Instead of using the term "fault-equivalence class" from Hari and Li to denote groupings of multiple DUECs, I will use the term *Fault-Similarity Class* (FSC) in this chapter to avoid confusion with DUECs (see Section 3.3.1.1), and to capture the following facts:

MISPREDICTIONS MAY OCCUR: One DUEC – the *pilot* – represents all the other DUEC members of a FSC. Due to the approximative nature of heuristics, one or more non-pilot members of a FSC can have a different experiment outcome than the pilot, making the word "equivalence" unwarranted.

EQUIVALENCE IS IN THE EYE OF THE BEHOLDER: The "equivalence" of two FI-experiment outcomes purely depends on the experimenter's definition. While for one type of FI campaign, any deviation from the golden run is a *failure*, for others a detailed differentiation into several outcome types (see, for example, Section 4.6) is important. Hence, multiple DUECs are *similar* only under the chosen failure model.

Although both approaches [HANR11, HANR12, HANR13, LT13] invent various complicated analysis techniques to combine multiple DUECs into larger groups, the general notion of *fault similarity* can be reduced to a simple insight:



```
   Assembler Source Code
loop:
# READ!, EAX += array[EDX]
addl array(,%edx,4),%eax
# increase loop count/index
addl $1,%edx
# check loop abort condition
cmpl $1000,%edx
jne loop
```

```
EIP    Dynamic Instruction         ESP    EBP    EDX    EAX
0x4711 addl array(,%edx,4),%eax    0xffc0 0xffe4 0x0000 0x8403
0x4716 addl $1,%edx                0xffc0 0xffe4 0x0000 0x84d4
0x4718 cmpl $1000,%edx             0xffc0 0xffe4 0x0001 0x84d4
0x471a jne 0x4711                  0xffc0 0xffe4 0x0001 0x84d4
0x4711 addl array(,%edx,4),%eax    0xffc0 0xffe4 0x0001 0x84d4
0x4716 addl $1,%edx                0xffc0 0xffe4 0x0001 0x8591
0x4718 cmpl $1000,%edx             0xffc0 0xffe4 0x0002 0x8591
0x471a jne 0x4711                  0xffc0 0xffe4 0x0002 0x8591
0x4711 addl array(,%edx,4),%eax    0xffc0 0xffe4 0x0002 0x8591
0x4716 addl $1,%edx                0xffc0 0xffe4 0x0002 0x8ce2
0x4718 cmpl $1000,%edx             0xffc0 0xffe4 0x0003 0x8ce2
0x471a jne 0x4711                  0xffc0 0xffe4 0x0003 0x8ce2
```

Figure 5.2: Short IA-32 assembler snippet (top) adding up the contents of an array: All memory reads in the dynamic execution (bottom) share a similar machine state, and FI will lead to similar results in all cases – a wrong sum. (For simplicity, registers carry dummy 16-bit values.) *Adapted from [SBS14].*

> *Assume that at one point in time t_1 during the workload run, the machine state is completely identical to the machine state at another point in time t_2. We conduct two FI experiments, one injecting a fault at time t_1 in memory address m, the other one at time t_2 injecting in the same memory address m. Clearly, both experiments will yield the same result.*

For example, if one FI experiment results in an SDC, the very same thing will happen in the other experiment.

Of course, in reality the machine state is never completely identical at two points in time t_1 and t_2 during a program run; even if the benchmark would enter an infinite loop, some parts of the machine – for example, a wall-clock timer – would be in a different state. Nevertheless, over the program's runtime, a *relevant part* of the machine state may be identical at several points in time.

To gain a better intuition on what is meant by a "relevant part" of the machine state, observe the IA-32 assembler code snippet in Figure 5.2 (upper half): In a loop, the integer elements of an array are added up in the EAX register, keeping the array index – also used for the loop-abort condition – in the EDX register. When considering faults in memory only, and applying the def/use pruning method described in Section 5.2.1, the only memory-reading *(use)* instruction is the one in line 2, marked with the comment `"READ!"`. Injecting a fault into the memory location being read from directly before the read will lead to a similar result in all loop iterations: The resulting sum will be calculated faultily. Thus, it would suffice to do a single experiment instead of 1000, and predict the same result outcome for the others.

Now consider the machine state right before each memory read in the dynamic execution (lower half of Figure 5.2, highlighted lines): Among others, the `EIP` (instruction pointer), `ESP` and `EBP` registers are the same in all cases, `EDX` only differs by its lower-order bits in most consecutive loop iterations, and `EAX` may – depending on the magnitude of the values in the array – not change too much from one iteration to the next either. A geometric *projection function* of the machine-state vector – preserving only the components `EIP`, `ESP`, `EBP`, and the higher-order bits of `EDX` and `EAX` – therefore serves very well as a criterion to combine all these DUECs into a single FSC, and to conduct a single experiment instead of one per loop iteration.

5.2.3 *A Flexible Fault-Similarity Heuristic*

The working hypothesis is that a *projection of the machine-state vector* can successfully be used to combine multiple DUECs to a FSC with high result accuracy, meaning that – depending on the user's requirements – the outcomes of most of the grouped DUECs are *similar*. I assume that this projection highly depends on several factors:

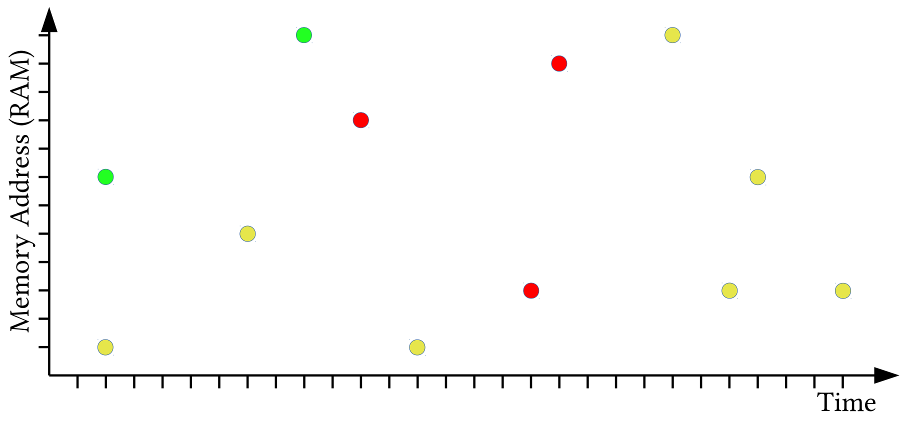
- Analyzed workload, including the underlying operating system, and its chosen input.
- Compiler, and chosen compiler optimizations.
- Machine model, including CPU architecture, and time discretization.
- Fault and failure model.

A generic fault-similarity heuristic therefore has to adapt to these factors without having to specialize the approach, for example, for one CPU architecture, like Hari et al.'s Relyzer [HANR11, HANR12, HANR13].

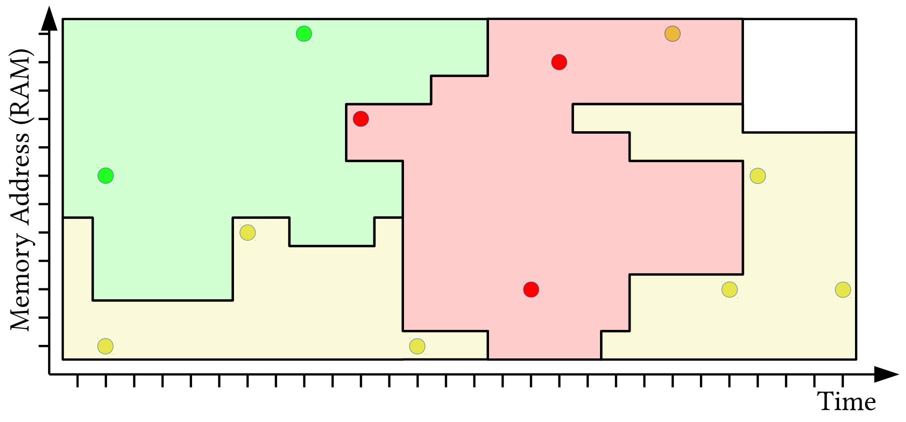
The basic idea behind the FSP heuristic is *to find a suitable machine-state projection* that can be used to accurately combine DUECs with "mostly" equivalent FI results. The search for this projection function is divided into a preparation step that records the machine-state vectors during the golden run, a sampling step that collects FI results as the training set, and the projection-function search step itself.

5.2.3.1 *Preparation: Record Machine-State Vectors*

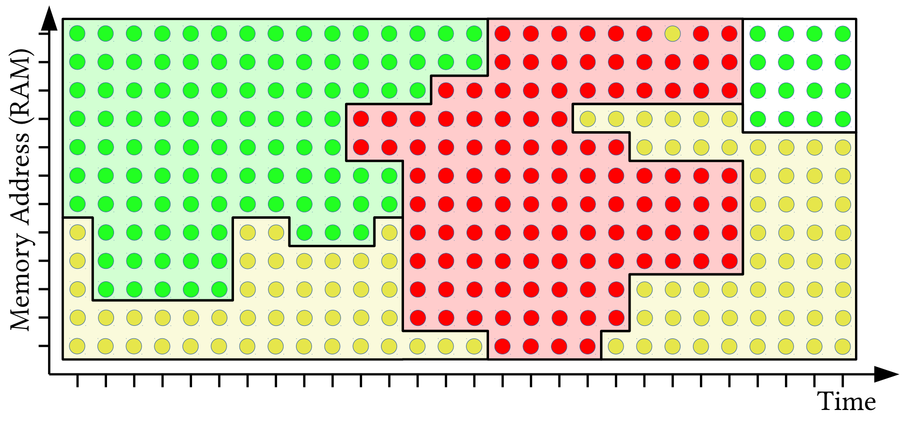
First, the approach records a machine-state vector for each *def* and *use* when recording the golden-run trace (see Section 3.1.2). For efficiency reasons, I do not record the *complete* machine state, but only the values listed in Table 5.1. Among other information, FSP records the current contents of all



(a) *Sampling:* In the first step of the FSP approach, we run FI experiments for a randomly-chosen, feasible subset of all DUECs.



(b) *Partitioning:* Each possible state-vector projection-function partitions the fault space into FSCs. In the example, the partitioning is not perfectly accurate, as the top-right yellow "timeout" FI result for one DUEC is grouped together with three red "SDC" results. For the white FSC in the top right, no FI experiment has been conducted yet.



(c) *Prediction:* After conducting one experiment for the top-right FSC (outcome: "No Effect"), the outcomes for the remaining DUECs can be predicted from the already obtained results.

Figure 5.3: FSP steps: Sampling, fault-space partitioning, and result prediction. Note that in this simplified illustration, each colored circle is meant to represent one DUEC, and each black-bordered polygon one FSC combining multiple DUECs.
Legend: *green* = No Effect, *yellow* = Timeout, *red* = SDC.

machine registers, the contents of the memory word each of them points at when interpreting their content as a memory address, the current machine-instruction's opcode, and the actual value the CPU reads from or writes to memory.

5.2.3.2 *Step 1: Sampling*

Then, FSP determines FI results by running actual experiments for a feasible, randomly chosen subset of all DUECs, as shown in Figure 5.3a. In the next step, these results will serve as the *training set* for searching a state-vector projection that accurately groups DUECs with mostly equivalent experiment outcomes to FSCs.

5.2.3.3 *Step 2: Projection-Function Search*

Based on the training set, an optimization algorithm then searches for a projection function that is (near-)optimal regarding a specified result accuracy, or a limit on the number of FI experiments – which refers to the number of FSCs, as the approach runs one FI experiment for each *pilot* DUEC. Together with the machine-state information for each *def* and *use* from the golden run, each possible projection function – for example, the one outlined in Section 5.2.2 – partitions the fault space into a set of FSCs, as shown in Figure 5.3b. DUECs that have the same projected machine-state vector are grouped into the same FSC. Speaking in the example from Section 5.2.2, all DUECs with the same values in `EIP`, `ESP`, `EBP`, and the higher-order bits of `EDX` and `EAX`, are assigned to the same FSC.

For some of these DUECs, the FSP heuristic already has determined an FI result in step 1 (see above).[1] The result *accuracy* measures how well the partitioning and the sampling FI results match. In fact, the two optimization criteria – *maximum accuracy* and *minimal number of FSCs* – are contradictory and can be traded for each other. The extremal points at each end of the solution space help understanding the trade-off:

- The extremal point in favor of *accuracy* uses the identity function as the state-vector projection. It defines *all* available machine state as *relevant* for grouping – and combines *no* DUEC with another: It produces FSCs with *one* DUEC member each. As the approach needs at least one real FI result per FSC, this solution requires an FI experiment for *every* DUEC, and achieves maximum accuracy.

- The other extremal point combines *all* DUECs to a single FSC and only conducts one single experiment. It results in minimal experimentation effort and maximal result error.

Between these extremal points exists a large solution continuum with all possible machine-state vector projections. Some of these solutions are Pareto-

1 For those FSCs we do not have at least one real FI result for yet, we will conduct an FI experiment in Section 5.2.4.

optimal [Deb01], which means they optimally trade experiment effort for accuracy.

5.2.4 *Applying the Similarity Heuristic*

After finding a projection function – and, hence, one specific fault-space partitioning scheme, – that satisfies the user's requirements regarding accuracy and FI-campaign efforts within the training set, the user can apply it to the remaining DUECs with yet unknown outcome.

Some FSCs do not yet contain at least one DUEC member with a known outcome, for example the white FSC in the top-right corner of Figure 5.3b. Because no FI experiment was run for these FSCs in the sampling step, the approach now picks one *pilot* DUEC for each of them, and runs the FI experiment for it.

After this step is completed, all DUECs, and, thus, every coordinate in the fault space either directly or indirectly can be assigned an experiment outcome: Directly, by running an FI experiment for them, or indirectly, by looking at the *pilot* in the FSC they belong to. Figure 5.3c illustrates this result-prediction step, after which the complete fault space can be "colored" with FI-result outcomes.

5.3 IMPLEMENTATION

I implemented the outlined FSP approach as an extension to FAIL* (see Chapter 4). I extended the *Tracing* plugin and the associated *Trace_Event* protocol-buffer message (see Section 4.5.2) with the capability to record the additional machine state listed in Table 5.1 alongside the usual instruction and memory-access trace. The `import-trace` (see Section 4.5.3.3) tool's new `--importer AdvancedMemoryImporter` and `--extended-trace` switches import the data into additional columns of the *trace* database table.

5.3.1 *Encoding Machine State and Projection Function*

The FSP approach itself is implemented in the `fail-eq-search` tool in about 2000 lines of code.[2] I encode the recorded machine-state listed in Table 5.1 into a single, long bit vector with all state variables concatenated. This *state vector* exists once for each dynamic instruction in the golden run that reads memory. The projection function the FSP heuristic want to search for is also encoded as a bit vector, which has the same length as the state vector, and is called the *projection vector*. Its bits indicate whether the corresponding bit position in the machine state is *used* (1) or *not used* (0) for comparing the machine state of DUECs, deciding whether or not to group them into a common FSC. The bitwise AND of an – initially randomly chosen – projection vector

2 When leaving its prototype status, the FSP implementation will become part of the `prune-trace` tool (see Section 4.5.3.4).

RECORDED ITEM	DESCRIPTION
data_address	Memory address the def/use writes/reads
data_value	Actual value that is written/read
EIP	Instruction pointer of the def/use instruction
dyn_instr	Dynamic instruction count since workload start
opcode	Instruction's opcode
EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP, EFLAGS	Contents of general-purpose registers, stack pointer, CPU flags
*EAX, *EBX, *ECX, *EDX, *ESI, *EDI, *ESP, *EBP	Contents of the machine word the respective register points to (if interpretable as a mapped memory address)
jumphistory	Relyzer [HANR11, HANR12, HANR13] style *control-equivalence* bit list indicating whether each of the last 16 and next 16 conditional branches were taken
duration	Temporal duration of the def/use equivalence class (in CPU cycles)
benchmark_id	An ID uniquely identifying each benchmark

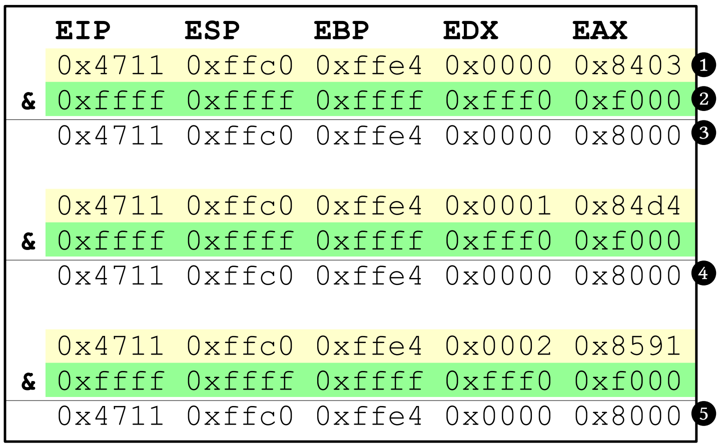
Table 5.1: Information recorded for every dynamic *def* or *use* instruction executed during the golden run of each workload.

and each state vector yields a *comparison vector* uniquely identifying the FSC each DUEC belongs to. The example projection function from Section 5.2.2 – preserving only the components EIP, ESP, EBP, and the higher-order bits of EDX and EAX – can directly be encoded by setting the corresponding bits in this vector, as demonstrated in Figure 5.4.

5.3.2 *A Genetic Algorithm for Searching Optimal Projections*

In order to find an good or near-optimal projection vector, I model the search problem on the training data using the SPEA2 multi-objective evolutionary algorithm [ZLT02] as implemented in the PISA library [BLTZ03] with the projection vector as the genome, and simple multi-bit mutation and single-point crossover operators [Mit98]. Initially I run a fixed number of 100 000 FI experiments[3] per benchmark to gain training data. Then the genetic algo-

3 This number was arbitrarily chosen, but may be selected application-specifically or depending on the available computing resources.



	EIP	ESP	EBP	EDX	EAX	
	0x4711	0xffc0	0xffe4	0x0000	0x8403	❶
&	0xffff	0xffff	0xffff	0xfff0	0xf000	❷
	0x4711	0xffc0	0xffe4	0x0000	0x8000	❸
	0x4711	0xffc0	0xffe4	0x0001	0x84d4	
&	0xffff	0xffff	0xffff	0xfff0	0xf000	
	0x4711	0xffc0	0xffe4	0x0000	0x8000	❹
	0x4711	0xffc0	0xffe4	0x0002	0x8591	
&	0xffff	0xffff	0xffff	0xfff0	0xf000	
	0x4711	0xffc0	0xffe4	0x0000	0x8000	❺

Figure 5.4: Calculation of the comparison vector (on the example from Figure 5.2): A bitwise AND of each *machine-state vector* ❶ and the *projection vector* ❷ yields the *comparison vector* ❸. DUECs with the same value for the comparison vector (❸, ❹, and ❺ are identical) are grouped in the same FSC, and only a single FI experiment – the *pilot* – represents all other DUEC members of the FSC.

rithm is initialized with a population of randomized projection vectors. In every *generation* of the search algorithm, each individual's – or, each projection bit vector's – fitness is evaluated by …

1. performing the aforementioned grouping of DUECs from the training set into FSCs,

2. picking the largest[4] DUEC in each FSC as the pilot and pretending it properly represents the remaining DUEC members of the FSC, and

3. measuring the two fitness criteria *accuracy* and the emerging *number of FSCs* within the training set.

The *accuracy* measures how accurately the pilots *actually* represent the remaining DUEC members within their FSCs. As a misprediction of a large – many clock-cycles wide – DUEC has a greater impact on the outcome quality than a small one, the correctly predicted *area* in the fault space is used for this metric, taking the weights of the DUECs into account:

$$\text{Accuracy} \quad = \quad \frac{\text{Correctly predicted fault-space area}}{\text{Total fault-space area}}$$

The second fitness criterion is the *number of different FSCs* emerging from the DUEC grouping step. These two criteria – the number of correctly represented faults, and the number of FSCs that directly translates into the number of pilots – are used as the optimization objectives for the SPEA2 algorithm.

4 Currently the def/use class spanning the most CPU cycles is chosen as each similarity class's pilot to minimize error when mispredictions occur.

BENCHMARK	DYN. INSTR.	CPU CYCLES	#EXP. AFTER DEF/USE PRUNING	FI SIM. RUNTIME
ECos/ BASELINE	1.08×10^7	9.7×10^9	1.48×10^7	4946 h (33.4 h
ECos/ CRC	2.73×10^7	9.7×10^9	4.15×10^7	15 035 h (36.1 h)
MiBench/ QSORT	4.20×10^7	4.20×10^7	1.48×10^7 (of 5.49×10^7)	44 139 h (297.6 h)
MiBench/ BASICMATH	1.47×10^8	1.47×10^8	1.24×10^7 (of 1.84×10^8)	95 068 h (767.8 h)
MiBench/ BITCOUNT	4.08×10^7	4.08×10^7	2.89×10^6 (of 9.15×10^6)	3621 h (124.9 h)
MiBench/ SUSAN	2.95×10^7	2.95×10^7	1.25×10^7 (of 3.68×10^7)	23 526 h (186.9 h)

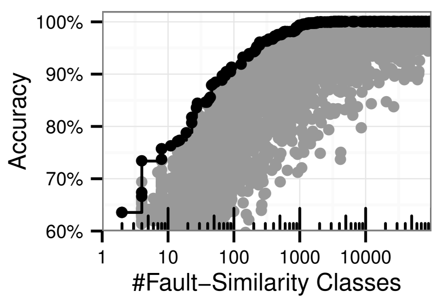
Table 5.2: Dynamic instruction counts, simulated CPU cycles, number of eight-bit burst FI experiments necessary after def/use pruning (see Section 3.3.1.1), and FI simulation runtime for all experiments (and the 100 000 experiment training set in parentheses). For the eCos benchmarks, the total CPU cycles differ from the dynamic instructions due to idle phases; for MiBench, we limited FI to the first 1×10^7 dynamic instructions.

5.4 EVALUATION

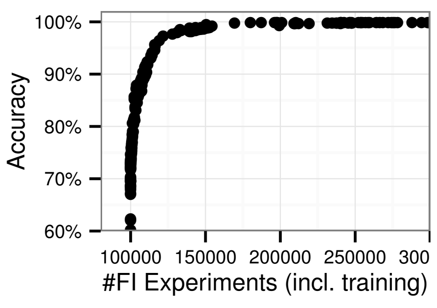
This section elaborates on the evaluation setup, and subsequently analyzes the effectiveness and efficiency of the FSP heuristic.

5.4.1 *Evaluation Setup and Ground Truth*

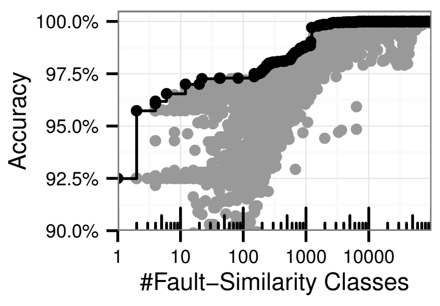
As a test workload, I chose the eCos kernel-test programs I already used in the GOP case study (see Section 4.6.2). The ECos/BASELINE programs are relatively small, and also reasonably similar to each other regarding their fault-propagation patterns, which allows this evaluation to consider them as a single, combined benchmark. A second variant of the test programs – ECos/CRC – is hardened against memory faults with the CRC variant of the GOP (see Section 4.6.4) and an additional protection for the stacks of preempted threads [HBD⁺14]. ECos/CRC executes about three times more dynamic instructions than ECos/BASELINE (cf. Table 5.2). Additionally, I picked MiBench's [GRE⁺01] *automotive* benchmark category as a set of real-world application benchmarks. The four benchmarks – QSORT, BASICMATH, BITCOUNT, SUSAN, using the *small* input data set – each execute more dynamic



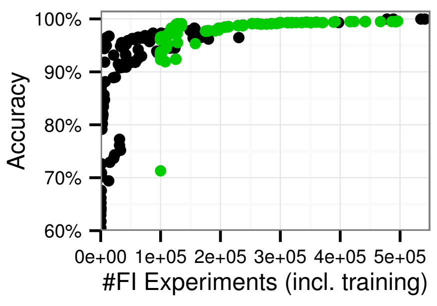
(a) eCos/baseline (training)



(b) eCos/baseline (total)



(c) eCos/CRC (training)



(d) eCos/CRC (total, w/ and w/o training)

Figure 5.5: Accuracy in the training set (left) and in the complete fault space (right) for eCos/baseline and eCos/CRC.

instructions than all eCos benchmarks combined, and represent a more heavyweight workload for our tooling.

To determine the "ground truth" for the pruning experiments, I ran FI experiments for *all* DUECs of these benchmarks, resulting in the total simulation time shown in the last column of Table 5.2.[5] In the following, I will use this data as a basis of comparison to rate the quality of the FSP method's results.

As described in Section 5.3.2, the `fail-eq-search` tool randomly[6] picks 100 000 DUECs for each benchmark in Table 5.2 as the training set. The genetic algorithm was parametrized with a population size of 100 individuals, 400 optimization generations, and a mutation probability of 10 percent. I picked the values for these parameters from experiences in early evaluation rounds.

5.4.2 *Heuristic Training and Test*

Figure 5.5a shows the training results and fitness values for eCos/baseline after 3:40 minutes of optimization on a 32-core Intel Xeon E5-4650: Each point represents an individual in terms of the genetic algorithm – each with a specific *machine-state projection vector* (see Section 5.2.3), which partitions

5 I limited FI to the first 10^7 dynamic instructions for the MiBench benchmarks to keep the computing time budgets reasonable.

6 Uniform sampling without taking the DUEC size into account.

the training set into a number of FSCs as shown on the log-scale X-axis with possible values from 1 to 100 000. As described in Section 5.3, the accuracy of such a partitioning – plotted on the Y-axis – is defined as the percentage of correctly represented fault-space area.

The optimization yields a set of Pareto-optimal solutions (black) the user can choose from to partition each workload's *complete* fault space in the next step (see Section 5.2.4). Figure 5.5b shows the test results when applying all previously determined, Pareto-optimal projection vectors to the complete fault space while reusing all results from the 100 000 training FI experiments: Depending on the partitioning the FSP heuristic creates in the complete fault space, more experiments than the initial training experiments need to be conducted to get a representing pilot for *all* new FSCs. For a chosen projection, the accuracy drops by a nonlinear factor – in the order of 0.1 percent for highly-accurate, up to 20 percent for the low-quality solutions – from training to test. For example, a solution with 99.9986 percent accuracy in the training set (9012 FSCs) requires the user to conduct 99 308 additional FI experiments, totaling in 199 308 including the training set, from a total of 14.8 million (see Table 5.2). This solution reconstructs the complete fault space with 99.2512 percent accuracy; a solution with 99.9903 percent training-set accuracy and 2100 FSCs yields 90.4036 percent accuracy with 8243 additional experiments (108 243 total).

In Figure 5.6, I apply the latter example solution to the complete eCos/baseline fault space, illustrating the advertised local fault-space feature preservation of the FSP heuristic: A close-up of a small area of the fault-space plot of the eCos/baseline mutex1 benchmark even remains largely intact when after training only 8243 additional FI experiments are conducted – totaling 108 243 experiments including the training set, a mere 0.73 percent of all 14.8 million eCos/baseline experiments.

Subsequently I investigated how well previously trained solutions apply to a new and unknown, yet not completely different workload. As described in the previous section, the eCos/CRC benchmark comprises the same kernel-test programs as eCos/baseline, yet they are hardened against memory faults, and execute substantially more dynamic instructions. Probably most notably they introduce a new FI-experiment outcome type "detected" that signals a successful error detection by the CRC-based EDM: This outcome type does not exist in eCos/baseline, and, thus, cannot have been observed by the training process from Figure 5.5a. The black points in Figure 5.5d show how well these projection vectors perform for eCos/CRC without any eCos/CRC-specific training. Note that without this training phase, there is no initial 100 000 FI-experiment penalty for the training set. One interesting observation is that the solutions requiring up to 300 000 FI experiments are partially in the 90 to 97 percent accuracy range, but by far not as close to 100 percent as in the eCos/baseline plot (Figure 5.5b). Nevertheless, previously trained projection vectors seem to be reusable even for unknown benchmarks: As the training process only learns how to group def/use classes into fault-similarity classes, but does not try to completely *predict*

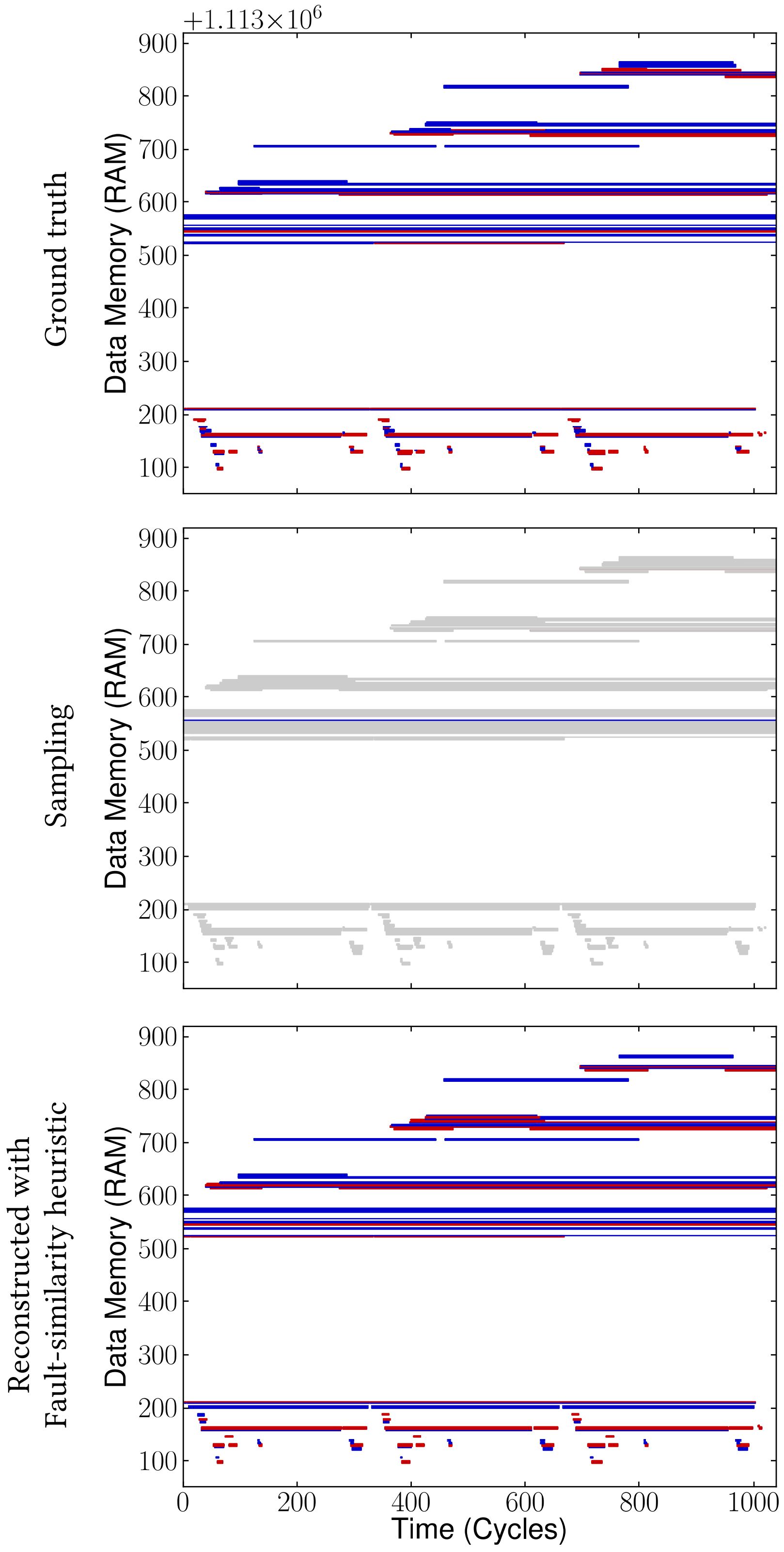


Figure 5.6: A tiny fault-space plot excerpt from a stack memory area of the MUTEX1 benchmark (from top to bottom; color coding: *white* = no effect, *blue* = timeout, *red* = CPU exception, *black* = SDC): Ground-truth results (100 % FI experiments), results from sampling 0.73 % of all DUECs (gray areas are unknown results, i.e., DUECs that were not sampled or known a priori), and reconstructed results with also a total of 0.73 % (including training set) of all experiments for the ECOS/BASELINE benchmarks. *Adapted from [SBS14].*

experiment outcomes without carrying out new FI experiments, it can even deal with previously unseen experiment outcomes, such as "detected" in this case. Figure 5.5c and the green points in Figure 5.5d show the accuracy results after training specifically for eCos/CRC:[7] The accuracy, and especially the accuracy mapping from training to test, is significantly better than without training – the low-quality left margin vanishes. This, of course, comes at the cost of an initial training phase and more FI experiments depending on the desired accuracy.

Among the remaining Figure 5.7a–5.7h, MiBench/qsort displays an extremely high accuracy even for minimal numbers of additional experiments: With 100 012 FI experiments – results from training, plus 12 new FSCs in the complete fault space – it achieves an accuracy of 99.9902 percent. The primary reason for this is that the vast majority of experiment outcomes – 99.9892 percent – for this benchmark are SDCs,[8] which allows to create extremely large FSCs that yield the same outcome.

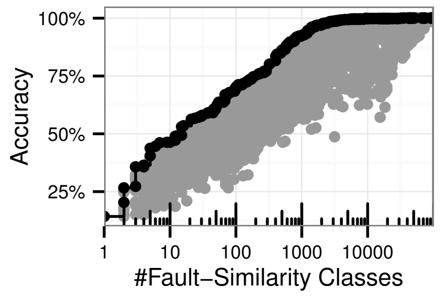
5.4.3 *Experiment-Outcome Breakup and Comparison with Sampling*

As the user is, apart from fault-space details, also interested in the usual experiment outcome breakup, this aggregate should not turn out to be inaccurate either. For example, in the basicmath benchmark, 34.18 percent of all faults result in *no effect*, 19.34 percent in an SDC, 31.80 percent in a *CPU exception*, and 14.68 percent in a *timeout*. Figure 5.8 shows the *Root Mean Squared Error* (RMSE) of experiment outcome breakups for a selection of benchmarks with the Pareto-optimal heuristics from Figure 5.5 applied to their complete fault space (black points and green points with the same meaning as in the previous section): As expected, the fault-space reconstruction accuracy and the outcome breakup RMSE correlate quite well – good local accuracy also yields a good global accuracy.

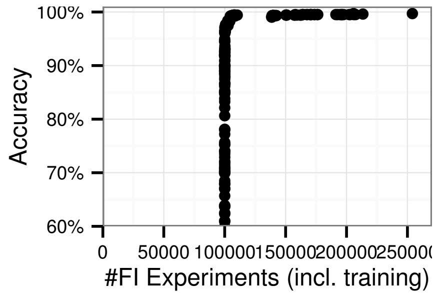
The plots in Figure 5.8 also include the outcome-breakup RMSE for FI sampling (with fault expansion [SJAP97]; red lines in the figure), a technique commonly used to *only* estimate the breakup without gaining any information on local fault-space details. Interestingly, in some cases sampling yields inferior results – especially for the un-trained heuristic configurations in the eCos/CRC case (Figure 5.8b, black points) as it does not have the experiment-count penalty of a training set. This means the FSP heuristic can compete with sampling, although it yields much more detailed information on the fault space.

7 … and reusing some of the eCos/baseline projection vectors as the initial population for the optimization algorithm.

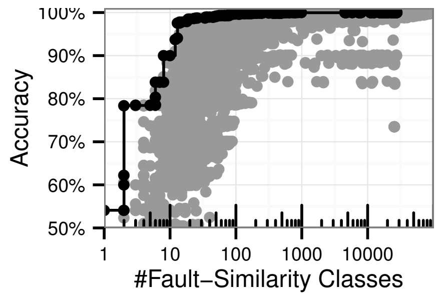
8 This failure behavior is in fact not very astonishing for a benchmark sorting a long list of text strings.



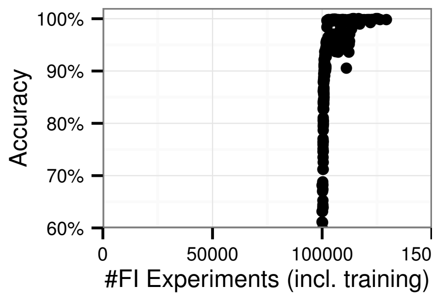
(a) MiBench/basicmath (training)



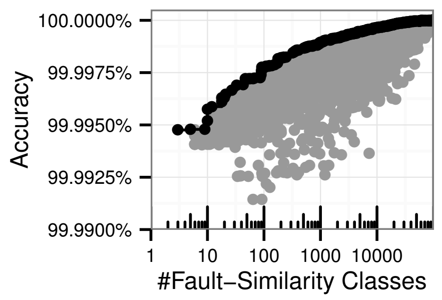
(b) MiBench/basicmath (total)



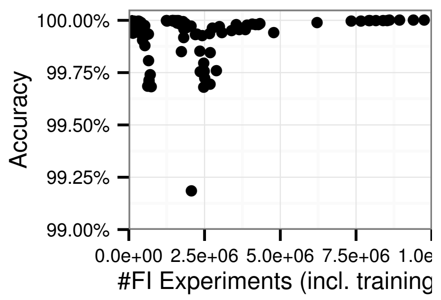
(c) MiBench/bitcount (training)



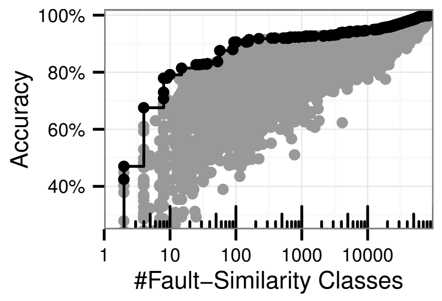
(d) MiBench/bitcount (total)



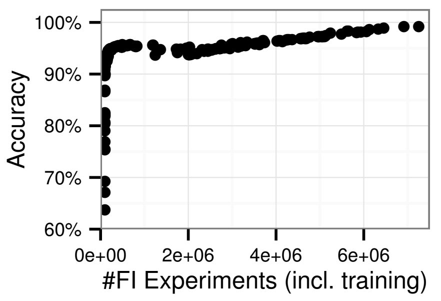
(e) MiBench/qsort (training)



(f) MiBench/qsort (total)

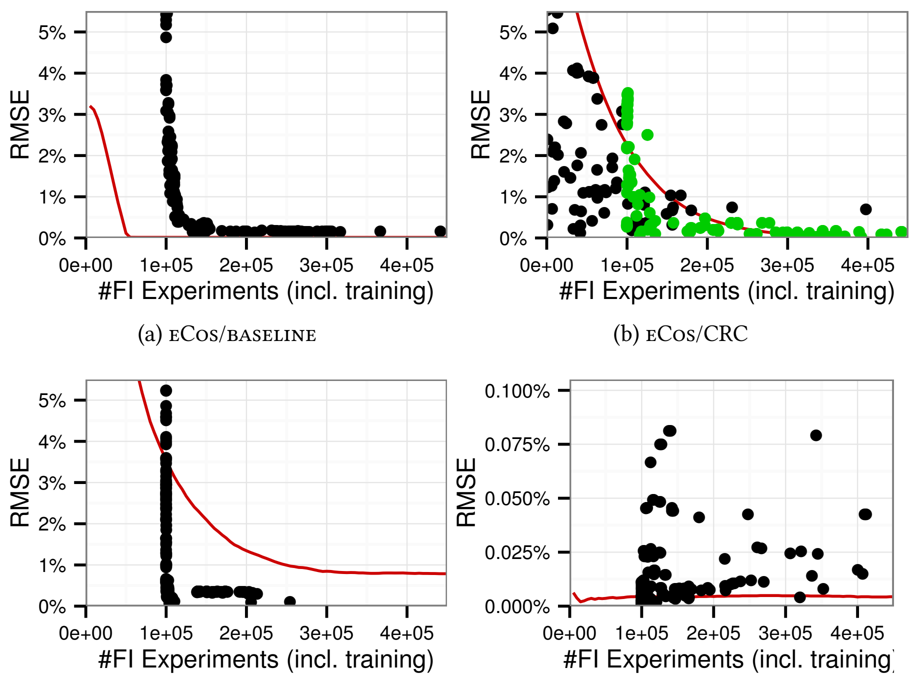


(g) MiBench/susan (training)



(h) MiBench/susan (total)

Figure 5.7: Accuracy in the training set (left) and in the complete fault space (right) for the four MiBench *automotive* benchmarks.



(a) eCos/baseline (b) eCos/CRC

(c) MiBench/basicmath (d) MiBench/qsort

Figure 5.8: Comparison to sampling (red line): RMSE of the outcome probability breakup is comparable to, and many cases better than, the common sampling approach – which completely lacks information on local fault-space details.

5.5 SUMMARY AND CONCLUSIONS

To summarize, in this chapter I presented an adaptive, application-specific fault-space pruning technique that preserves local features of the fault space for detailed susceptibility analyses. The *Fault-Similarity Pruning* (FSP) approach – an extension to Fail*– allows the user to freely trade accuracy for experimentation runtime, choosing a Pareto-optimal heuristic configuration that was trained with a feasible FI-experiment subset. The evaluation results confirm the assumption that a machine-state subset can be successfully used to partition the fault space into FSCs, allowing to gain insights on local phenomena for EDM/ERM placement with massively reduced experimentation efforts: For example, when the user chooses to run 1.5 percent of all FI experiments, the average[9] result accuracy is 99.84 percent.[10] In many cases the FSP technique even outperforms classic sampling techniques, although they do not preserve any fault-space details.

9 The average weighted by the total number of faults in each benchmark.

10 Except for MiBench/bitcount, where training yields no solution that only needs 1.5 percent of all experiments. Here, the user can achieve, for example, 99.84 percent accuracy for 4 percent of all experiments.

6 SMART-HOPPING: CAMPAIGN SPEEDUP BY SINGLE-EXPERIMENT ACCELERATION

"We can chart our future clearly and wisely only when we know the path which has led to the present."

— Adlai Ewing Stevenson II, US politician, 1952

Contents

The primary performance bottleneck of test-access port based FI, as used in the JTAG-based FailPanda back end for Fail* (see Section 4.3.3), is the *fast-forwarding operation* that deterministically executes the target machine code without FI until a specific dynamic instruction is reached. At this point in the instruction stream – which is also present in the previously recorded "golden run" trace and identified by its position in that trace, – the execution must be stopped to inject the next fault (see Section 3.1.2).

Unfortunately, most embedded CPUs do not support this operation efficiently: Without further information, the target platform must single-step instructions until the specified instruction is reached. The widely used JTAG debugging interface [MT90], especially when controlled through low-cost

USB debugger hardware, incurs significant round-trip times between each step, as already demonstrated in Section 4.3.3.4. This results in extremely long experiment runtimes – dominated by the fast-forwarding operation, – and consequently a low FI experiment throughput endangering sufficient fault-space coverage.

After recapitulating existing solutions for fast-forwarding (Section 6.1), I present an approach that significantly speeds up this operation for most workloads with minimal requirements on hardware support (Section 6.2). Exploiting the information from a previously recorded instruction trace – which is needed anyways for systematic FI experiment planning and fault-space pruning (see Section 3.3 and Chapter 5), – the approach uses standard debugging hardware (breakpoints, memory watchpoints) to advance to a chosen point in the program execution with a minimal number of steps. The approach was implemented as an extension to Fail* respectively FailPanda (Section 6.3). The evaluation with MiBench's [GRE+01] *automotive* and *network* benchmark categories shows an improvement in experiment throughput by up to several magnitudes compared to similar FI tools in the field (Section 6.4). I discuss the evaluation results extensions to the approach in Section 6.5, and conclude in Section 6.6.

Parts of this chapter were originally published on ETS 2014 [SRS14].

6.1 FAST-FORWARDING: STATE OF THE ART

The problem of fast-forwarding to a specific dynamic instruction of a deterministically executing program has been studied not only in the FI domain, but is known especially in the context of debugging under the concept of "deterministic replay". For example, King et al. [KDC05] leverage checkpoints and Intel's hardware performance counters to deterministically re-execute parts of the target to create an illusion of "time travel" backwards and forwards through the execution of a program. More recently, Patil et al. [PPS+10] extended a similar approach to multi-core replay without any specific hardware support requirements, but at the expense of slow execution in a virtual-machine environment with JIT compilation. In his *Jockey* [Sai05] tool, Saito uses binary patching techniques and user-defined program annotations that are impossible to use for advancing to arbitrary instructions.

The existing approaches are too slow to run on embedded hardware, modify the target software to a point where the *observer effect* becomes relevant, or depend on special hardware features such as cycle-accurate performance counters. The following sections outline the fast-forwarding approaches currently predominant in the FI domain.

6.1.1 *Single-Stepping*

The most straightforward approach to advance to the N^{th} dynamic instruction is *single-stepping N times*. Benso et al. use single-stepping in their EXFI SWIFI tool [BPRSR98a, BPRSR98b], based on the *Trace Exception Mode* of

```
for (i = 0; i < 64; ++i) {
    for (j = 0; j < 64; ++j) {
        sum += array[i+j];
}   }
```

```
 8: b    30 <func+0x30> ; (outer loop entry)
 c: add  r12, r0, r3    ; i+j
10: ldrb r12, [r12, r2] ; array[...]
14: add  r2, r2, #1     ; ++j
18: cmp  r2, #64        ; j < 64
1c: add  r1, r1, r12    ; sum += ...
20: bne  c <func+0xc>   ; (inner loop back edge)
24: add  r3, r3, #1     ; ++i
28: cmp  r3, #64        ; i < 64
2c: beq  38 <func+0x38> ; (outer loop exit)
30: mov  r2, #0         ; j = 0
34: b    c <func+0xc>   ; (outer loop back edge)
```

Figure 6.1: Example ARM machine code (initialization omitted) and the C-code snippet it was compiled from: The highlighted *static* instruction at address `0x10` appears $64^2 = 4096$ times in the *dynamic* instruction stream. *Adapted from [SRS14].*

the Motorola processor in their prototype implementation. Due to the SWIFI technique (see Section 3.2.3), no host-PC/debugger/target-device round trip was involved. In contrast, in a TAP-based setting like FailPanda (see Section 4.3.3), this approach is feasible only for very short target programs: It cannot be used for reasonably sized workloads, as fast-forwarding to the dynamic instruction N grows linearly:

$$t_{\text{ff}} = N \times t_{\text{singlestep}} + t_{\text{exec}}$$

In this formula, t_{exec} is a negligible amount of time to actually execute the stepped instructions. For example, with $t_{\text{singlestep}} = 70\,\text{ms}$, fast-forwarding to the highlighted instruction in the second iteration of the inner and the second iteration of the outer loop in Figure 6.1 ($i = 1, j = 1$; $N = 400$) takes about 28.0 s – for a single experiment out of potentially thousands – on the PandaBoard, connected through a USB JTAG debugger (see Section 4.3.3.4). In this scenario, the actual – and indeed negligible – instruction-execution time can be approximated to

$$t_{\text{exec}} = \frac{N}{f_{\text{CPU}}} = \frac{400}{1.2\,\text{GHz}} \approx 333\,\text{ns},$$

assuming the PandaBoard's $f_{\text{CPU}} = 1.2\,\text{GHz}$ RISC CPU to execute without memory wait-states.

6.1.2 *Simple-Hopping with Hardware Breakpoints*

A more sophisticated approach is to use hardware breakpoints to go forward in the instruction stream by multiple dynamic instructions at once. Fidalgo

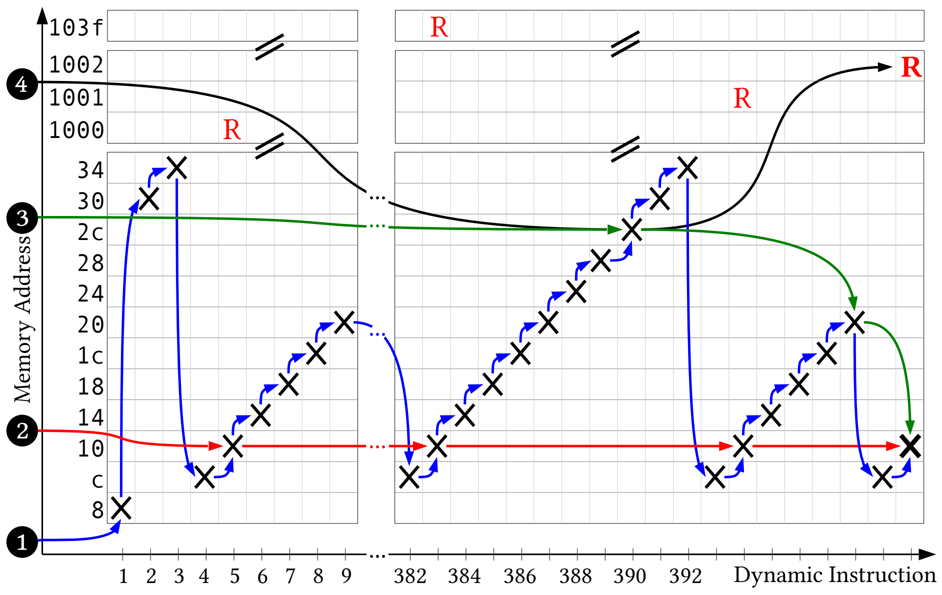


Figure 6.2: Pre-recorded "golden run" program trace with dynamic instructions (X = e**X**ecuted memory addresses), memory accesses (R = memory read), and different fast-forwarding methods: ❶ *Single-stepping*, ❷ *simple-hopping*, and ❸/❹ *smart-hopping* without/with memory watchpoint usage. *Adapted from [SRS14].*

et al. [FGAF06] set a breakpoint to advance to the *first* instance of a static instruction – this would correspond to the instruction `0x10` in Figure 6.1 (line 3), for the first time executed at position #5 in Figure 6.2 – but do not discuss the general case of reaching an arbitrary dynamic instance. A dynamic instruction with the property that it represents the *first* occurrence of a static instruction can therefore be reached with a single breakpoint hop. Unfortunately, this property only holds for a infinitesimal fraction of the dynamic instructions in real-world programs with loops and branches.

Prinetto et al. [PRSR98], Rebaudengo and Sonza Reorda [RSR99], Benso et al. [BRSR99], Folkesson et al. [FSK98], Barbosa and Skarin et al. [BVFK05b, SBK10], Cinque et al. [CCDM+09], Skarin et al. [SBK10, Ska10], and Hannius and Karlsson [HK12] also break on the static target instruction, but continue running the program after each breakpoint trigger *repeatedly* until the dynamic target instruction is reached. Figure 6.2 illustrates this *simple-hopping* procedure with the jump chain labeled with ❷: A breakpoint on the static instruction at address `0x10` triggers $N = 66$ times until the dynamic instruction #400 is reached. The fast-forwarding time can be calculated as

$$t_{\text{ff}} = t_{\text{exec}} + O \times t_{\text{BP}},$$

with O being the number of breakpoint triggers.

Assuming that a breakpoint trigger and a subsequent *continue* takes about the same time as a single-step, and t_{exec} is negligible, the time needed for fast-forwarding of this example is reduced by a factor of 6 compared to single-stepping. This reduces the actual time overhead to about 4.6 s on the PandaBoard ARM platform. However, Rebaudengo and Sonza Reorda note that in their BDM-based FI setup, the approach is still 71 to 97 times slower than the fault-free run, highly depending on the workload characteristics:

> *"Moreover, the required time is proportional to the number of activated breakpoints, because each activation involves the exchange of some information between the target microcontroller and the host PC through the slow serial interface."* [RSR99]

6.1.3 *Checkpointing*

The usage of checkpoints (see Section 3.3.2.1), which represent loadable intermediate states at a priori chosen – and usually equidistant – dynamic instructions $N_1, N_2, \ldots$, allows fast-forwarding to specific points in the execution trace. A checkpoint load costs a non-negligible amount of time, but more importantly *creating and storing* checkpoints consumes both time and storage space. Therefore, the total number of checkpoints in use is limited, making this technique orthogonal to fine-grained fast-forwarding techniques such as the aforementioned single-stepping or simple-hopping to navigate to the desired target instruction after the checkpoint load.

6.2 SMART-HOPPING: CHOOSING THE SHORTEST PATH

Unlike deterministic replay (see Section 6.1), deterministic fault injection has the advantage of a complete instruction and memory access trace that is usually recorded for experiment planning purposes. Extending on the simple-hopping approach explained in the previous section, I propose a fast-forwarding method that exploits more trace knowledge and forwards to a specified dynamic instruction with a *minimal* number of breakpoint hops.

Instead of (simple-)hopping from one dynamic instance of the target instruction to the next, the basic idea of the *smart-hopping* approach is to iteratively set a breakpoint on the static instruction that *will trigger farthest in the future* in the instruction trace. In the example in Figure 6.1 and Figure 6.2, this reduces the number of breakpoint hops and debugger round-trips to three (variant ❸ in Figure 6.2): One at static instruction `0x2c` (dynamic instruction #390), completely skipping the first iteration of the outer loop construct, one at `0x20` (#398), and one at the target instruction `0x10` (#400). On the PandaBoard ARM hardware, fast-forwarding time is reduced to about 0.21 s.

6.2.1 *Embracing Memory Watchpoints*

Although choosing an optimal breakpoint path significantly improves the example of reaching target instruction #400, this method degenerates when a late iteration of an instruction-wise homogeneous loop – with no variation regarding branch decisions in the loop body – contains the dynamic target instruction. For example, in a scenario trying to reach instruction `0x10` in

Algorithm 6.1 The *smart-hopping* algorithm generates the optimal jump sequence for all possible trace positions, reusing the solution sequence for the previous position. *Adapted from [SRS14].*

```
# Possible memory ACCESS TYPES in the trace:
type : enum AccessType { execute, read, write }
# Multiple Accesses may occur at one trace pos. (point in time):
type : class Access { Int address, AccessType type }
# Always holds the optimal hop sequence to current pos. in trace:
var cur_solution : list of tuples (Int trace_pos, Access a)
# Remember most recent occurrence of specific Access in trace:
var access_last_seen : map Access → Int trace_pos
# All memory Accesses at current trace position:
var cur_trace_events : list of type Access

cur_trace_pos ← 0
while not at trace end do
    read trace events at cur_trace_pos into cur_trace_events
    if any Access a ∈ cur_trace_events not in access_last_seen then
        clear cur_solution, add (cur_trace_pos, a) to it
    else
        new_hop ←
         {$x|x \in$ cur_trace_events ∧ access_last_seen[$x$] minimal }
        last_seen ← access_last_seen[new_hop]
        while length of cur_solution is > 1 do
            (pos$_a$, $a$) ← rightmost entry in cur_solution
            (pos$_{a-1}$, $a-1$) ← 2nd to rightmost entry in cur_solution
            if last_seen ≤ pos$_{a-1}$ then
                remove (pos$_a$, a) from cur_solution
            else
                break
            end if
        end while
        add (cur_trace_pos, new_hop) to cur_solution
    end if
    print cur_solution
    for all Access $x$ ∈ cur_trace_events do
        access_last_seen[$x$] ← cur_trace_pos
    end for
    cur_trace_pos ← cur_trace_pos + 1
end while
```

the last iteration of the inner loop ($i = 0, j = 63$ in Figure 6.1), the optimal breakpoint path contains 64 hops – yielding a fast-forwarding time identical to using simple-hopping.

Fortunately, standard OCD hardware offers another feature exploitable for fast-forwarding purposes: memory-access watchpoints. Similar to breakpoints, watchpoints trigger when an instruction accesses memory, and can usually even differentiate between read and write accesses. In the aforementioned pathologic example of the last iteration of the inner loop, the memory load instruction `0x10` accesses array element 63 exactly at the dynamic target instruction #400. This access can therefore be used as a watchpoint hop to directly forward to the desired target instruction, reducing the path length to *one* even for this scenario. Scenario ❹ in Figure 6.2 is further reduced to two hops.

Algorithm 6.1 outlines the smart-hopping heuristic that generates the optimal breakpoint/watchpoint hop sequence for *all* possible trace positions. It iteratively consumes dynamic instructions – including the memory accesses they perform – from a sequentially read instruction and memory-access trace, and reuses the – initially empty – solution hop sequence for the previous position. Although a solution sequence can grow to arbitrary lengths for input traces with adverse event patterns – such as homogeneous loops with extremely many iterations and no memory accesses (see Section 6.4), – the algorithm can be implemented with modest memory footprint even for very large input traces.

6.2.2 *Optimality*

For homogeneous hop costs – identical timing properties for breakpoints and watchpoints at arbitrary addresses, – the greedy smart-hopping heuristic is optimal due to an intuitive argument: *Not* picking the farthest-possible hop from the set of all reachable destinations cannot be better than picking the farthest, because the set of new potential hops reachable from a closer hop is always a subset of those reachable from the farthest-possible one. Consequently, choosing farthest hop target is always at least as good as choosing a closer target.

In case the hop costs are inhomogeneous – for example with different timings for breakpoints and watchpoints, or for breakpoints at the same instruction address as the current instruction pointer, – the generic solution would involve a path search. In fact, the evaluation in this chapter initially compared the smart-hopping heuristic to solutions generated by Dijkstra's path-search algorithm [Dij59], until I realized that smart-hopping was already optimal for homogeneous edge costs.

6.3 IMPLEMENTATION

The approach is implemented in two separate instances: as a separate evaluation tool for measuring the advantages of smart-hopping over simple-hopping and single-stepping, and as an extension to the *Database-Campaign* in FAIL*'s assessment-cycle layer (see Section 4.5.2.3) for speeding up concrete FI campaigns with FAILPanda (see Section 4.3.3).

6.3.1 *Hop-Chain Calculator*

For the purpose of evaluation and comparison in this chapter, the simple-hopping and smart-hopping heuristics were implemented in FAIL*'s `compute-hops` tool in about 700 lines of code. It takes an instruction and memory-access trace as an input, and calculates fast-forwarding breakpoint – and watchpoint, if enabled – hop chains for *every* dynamic instruction in the trace. Additionally, the forwarding costs are calculated.

Note that an actual FI campaign would not run an experiment for *every* dynamic instruction but carefully select a subset; nevertheless the calculated maximum and average costs are a good indication on how much faster the campaign gets with the different heuristics.

6.3.2 *Database-Campaign Extension*

To perform actual FI campaigns with the PandaBoard, FAIL*'s *Database-Campaign* and the accompanying *DatabaseCampaignMessage* definition were extended with the components from the `compute-hops` tool necessary for smart-hopping. Configurable at compile-time[1], the *Database-Campaign* optionally calculates the smart-hopping hop chain for each FI job it distributes, and embeds it into an extended version of the protocol-buffer message.

However, the smart-hopping algorithm required a small modification for the FAILPanda setup. Beyond the basic-operation measurements of the PandaBoard presented in Section 4.3.3.4, the JTAG-based TAP setup exhibited one subtlety that voids the algorithm's assumption of a single t_{BP} and t_{WP} value for the breakpoint respectively watchpoint timing: While a single-step and a normal hop between breakpoints and watchpoints take about 70 ms, hopping from a breakpoint to another breakpoint *on the same static instruction* – similarly with watchpoints – takes about *twice* that time (135 ms). The reason is that the debugger implements this case by deleting the breakpoint, stepping a single instruction, and setting the breakpoint again. Without the breakpoint deletion, the ARM processor would not continue execution but trap instantaneously – deletion and the single-step are not necessary if the new breakpoint is at a different address.

Subsequently, Algorithm 6.1 was slightly modified, avoiding the expensive hops by taking the *second-to-farthest hop* in these cases if possible, while still yielding optimal results when assuming a cost-difference factor of exactly 2. This change was also incorporated into the `compute-hops` tool, so that the evaluation in the next section reflects this detail.

1 Smart-hopping is enabled through the `CONFIG_INJECTIONPOINT_HOPS` CMake configuration switch, see Section 4.3.2.1.

6.4 EVALUATION

This section first presents the benchmark setup, and summarizes basic efficiency measurements of the `compute-hops` hop-chain calculation tool. The effectiveness evaluation of the smart-hopping approach in Section 6.4.3 applies the technique to real-world benchmarks, and compares it with the most advanced approach from the related-work section – *simple-hopping* (see Section 6.1.2).

6.4.1 *Benchmark Setup*

In order to evaluate the effectiveness of the fast-forwarding approach, I chose MiBench's [GRE+01] *automotive* and *network* benchmarks as a source for instruction and memory access traces. I compiled them with ARM-GCC 4.6.1 (`-march=armv5te`) and ran them in the gem5 simulator [BBB+11] in ARM system-call emulation mode to generate the traces for both the "small" and "large" parameter sets. The left quarter of Table 6.1 shows the benchmarks and their dynamic instruction and memory-access counts for the two parameter sets.

Note that for the purpose of this evaluation, the benchmarks were not actually run on the PandaBoard hardware itself, because only the fast-forwarding overhead – which can be calculated from the heuristic's output and the measured basic costs for the platform – is relevant for this chapter.[2]

6.4.2 *Hop-Chain Calculation Efficiency*

The `compute-hops` tool with enabled watchpoints processes about 1.5–2.3 million dynamic instructions per second – depending on the particular trace – on an Intel Xeon X5470 machine at 3.33 GHz, and its memory consumption peaks at 11.3 MiB when analyzing the "large" BITCOUNT trace. In essence, the hop-chain calculation is efficient enough even for large workloads, and can be used "online" – for example, as part of the *Database-Campaign* server – to generate hop chains for a running FI campaign.

6.4.3 *Smart-Hopping Effectiveness*

The table in Table 6.1 shows the maximum and average fast-forwarding costs[3]) for the chosen MiBench benchmarks. The results can be summarized as follows:

2 An actual FI campaign with FAILPanda and a reduced variant of the DIJKSTRA benchmark showed that the real fast-forwarding costs slightly exceed the theoretical values from this section. The reason is that t_{exec} is not negligible (see Section 6.1.1) when DIJKSTRA prints output to the serial port at runtime. This phenomenon, however, is in fact independent from the fast-forwarding operation itself.

3 Multiply by 70 ms for the actual fast-forwarding time (see Section 4.3.3.4).

MIBENCH BENCHMARK PROPERTIES ON ARMV5					SIMPLE-HOPPING			SMART-HOPPING (BP ONLY)			SMART-HOPPING (BP+WP)		
	BENCHMARK	INPUT	DYN. INSTR.	MEM. ACCESSES	max	avg	σ	max	avg	σ	max	avg	σ
automotive	BASICMATH	large	3.37×10^9	1.05×10^9	1.76×10^7	3.62×10^6	4.00×10^6	12 889	215.2	1246.2	8367	114.5	702.9
		small	2.66×10^8	5.00×10^7	2.92×10^6	7.21×10^5	7.55×10^5	267	10.5	12.3	177	4.7	7.3
	BITCOUNT	large	6.26×10^8	4.28×10^7	6.75×10^7	2.30×10^7	2.08×10^7	301 372	150 200.6	86 956.6	301 372	124 995.0	92 278.2
		small	4.18×10^7	2.86×10^6	4.50×10^6	1.53×10^6	1.38×10^6	20 122	10 051.3	5807.5	20 122	8374.9	6170.8
	QSORT	large	4.59×10^8	1.50×10^8	3.14×10^6	6.78×10^5	7.06×10^5	59 562	3203.7	4838.8	5	3.1	0.7
		small	1.78×10^7	7.12×10^6	4.63×10^5	6.73×10^4	8.85×10^4	6318	708.9	879.8	8	2.7	0.7
	SUSAN	large	3.92×10^8	1.52×10^8	9.95×10^6	2.54×10^6	2.93×10^6	28 849	116.0	689.5	40	2.9	1.0
		small	2.42×10^7	8.94×10^6	6.50×10^5	1.74×10^5	1.93×10^5	1883	55.3	57.5	40	2.8	0.8
network	DIJKSTRA	large	2.03×10^8	6.94×10^7	1.85×10^7	6.76×10^6	5.11×10^6	534	179.8	96.3	9	4.6	1.0
		small	4.59×10^7	1.60×10^7	3.67×10^6	1.18×10^6	1.04×10^6	534	153.3	103.2	7	3.8	1.1
	PATRICIA	large	5.81×10^8	2.25×10^8	5.10×10^6	4.47×10^5	6.13×10^5	70	9.4	2.7	8	2.7	0.7
		small	9.45×10^7	3.65×10^7	7.84×10^5	6.84×10^4	9.30×10^4	70	7.9	2.5	8	2.6	0.7

Table 6.1: Benchmarks from MiBench's *automotive* and *network* benchmark categories (with both *small* and *large* input data sets) with dynamic instruction counts on ARMv5 and memory access counts, fed through the *simple-hopping* heuristic, *smart-hopping* with only breakpoints, and *smart-hopping* with both break- and watchpoints. The results include maximum and average costs for fast-forwarding to *every dynamic instruction* in the respective benchmark. *Adapted from [SRS14].*

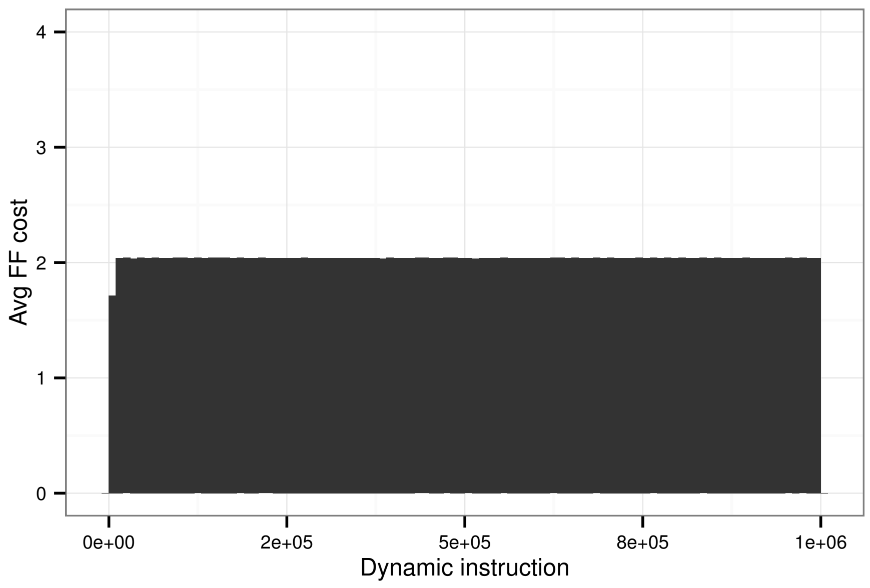
- The commonly used simple-hopping heuristic is clearly not usable for programs of this size: even for the "small" input of QSORT it takes an average of 78 minutes to fast-forward to a chosen dynamic instruction, and the other benchmarks are worse by a factor of up to 342.
- Using the smart-hopping heuristic without watchpoints already improves the situation significantly: Most benchmarks can on average be fast-forwarded within a matter of seconds, although the maximum forwarding costs are still relatively high, especially for the "large" inputs. The average fast-forwarding time improvement ranges between a factor of 94.9 for the "small" QSORT benchmark and 68 945 for the "small" BASICMATH. However, the BITCOUNT benchmark's costs stand out: On average it still takes 175 minutes to fast-forward the "large" variant, the maximum takes even twice that long.
- Allowing smart-hopping to additionally use watchpoints as hop targets again dramatically improves fast-forwarding times for almost all benchmarks: The average fast-forward is below 330 ms for all but BITCOUNT and the "large" BASICMATH, and also the maximum costs are within reasonable bounds. The improvement compared to the simple-hopping heuristic now ranges between 183.2 for the "small" BITCOUNT to 1.48 million for the "large" DIJKSTRA benchmark. Unfortunately, the problematic BITCOUNT benchmark still has extremely high forwarding times – its maximum costs did not improve at all with the help of watchpoints.

Figure 6.3 shows the average fast-forwarding costs for the first 10^6 dynamic instructions of the DIJKSTRA and the problematic BITCOUNT benchmarks. DIJKSTRA's trace constantly offers trace features that ease fast-forwarding, such as static instructions that are rarely executed, or specific memory accesses. In contrast, the costs for forwarding in the BITCOUNT benchmark linearly increase with the dynamic instruction count.

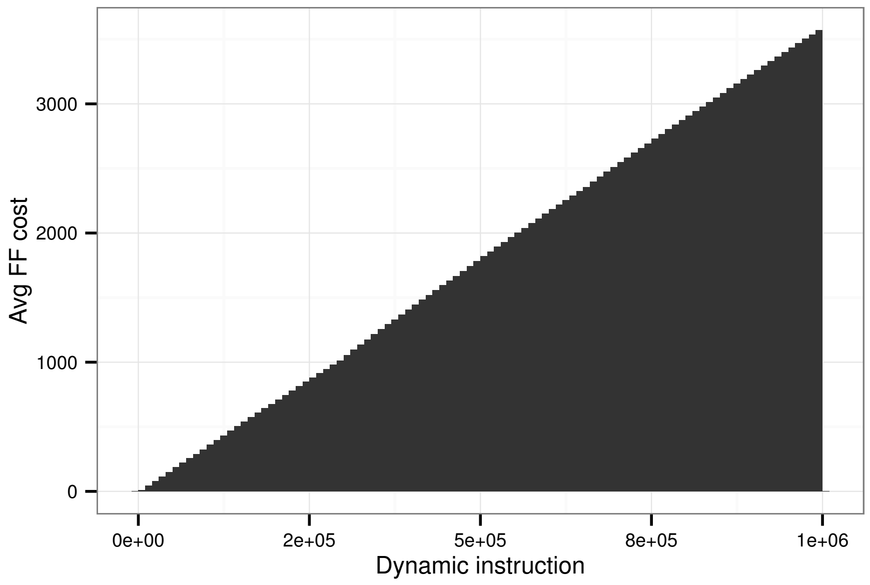
A closer investigation of the C code behind BITCOUNT reveals that its nested loop constructs repeat almost identical operations 1.125 million times in the "large" setup. Only the use of a pseudo-random number generator seems to introduce some variation. I believe that this pathologic setting is relatively unrealistic for the application domain, as FI experiments will not yield any new and interesting results after analyzing the first few of the million outer loop iterations.

6.5 DISCUSSION

The effectiveness evaluation showed that the smart-hopping heuristic improves fast-forwarding times massively for most workloads, as the round-trip time bottleneck between host PC and target platform dominates the total experiment time. For example, even for the shortest benchmark (QSORT



(a) dijkstra



(b) bitcount

Figure 6.3: Average fast-forwarding costs (bin size: 10^4 dynamic instructions) for the first 10^6 dynamic instructions of the "small" dijkstra and bitcount benchmarks.

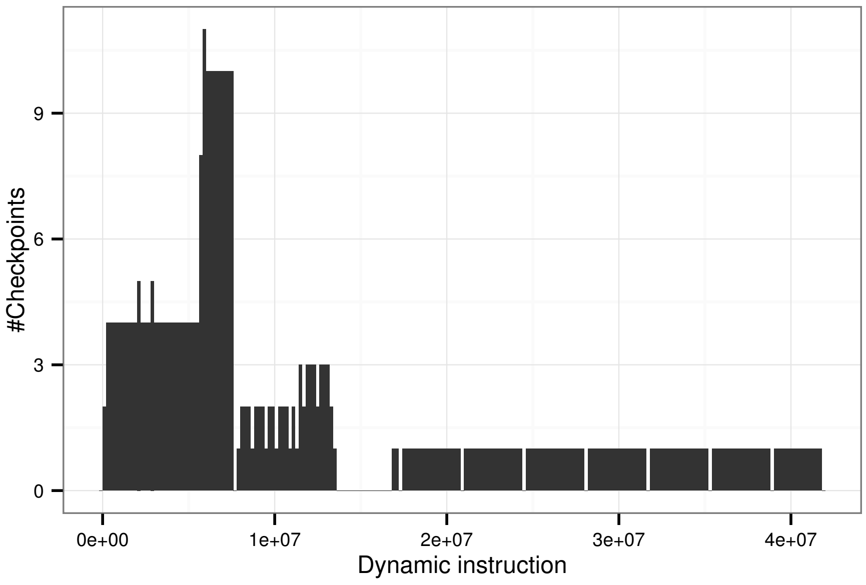


Figure 6.4: Distribution of 387 checkpoints (histogram bin size: 2×10^5 dynamic instructions) for the "small" BITCOUNT benchmark with a cost-capping checkpoint threshold of 500.

"small") the FI campaign throughput[4] improves from 18.3 experiments per day to 1707 without or 72 666 with watchpoints using smart-hopping on a single PandaBoard. This results in a fault-space coverage improvement of more than three orders of magnitude, with even more extreme improvements for longer-running programs – for example, DIJKSTRA "large" improves throughput by a factor of 357 943.

On the other hand, the BITCOUNT benchmark demonstrates the limitations of smart-hopping: In the worst case, even optimal hop chains are too long on average for reasonable fast-forwarding times. Programs with very long loops that only operate on register contents, or on a very small set of data from memory, cannot be efficiently fast-forwarded using hardware breakpoints and watchpoints only. In the remainder of this section I discuss an extension that would allow to handle these pathological cases as well. Additionally, I explore further optimization potential that factors in additional information from def/use pruning.

6.5.1 *Extending the Approach: Checkpointing*

For target programs that exceed reasonable average fast-forwarding costs, the approach can be complemented with *checkpointing* (see Section 3.3.2.1 and Section 6.1.3). The heuristic was extended by an automatism that creates a checkpoint whenever the costs amounting in the `current_solution` variable (Algorithm 6.1) cross a specified threshold, effectively capping fast-forwarding costs there. This threshold must be chosen large enough that

4 Assuming that setup/reset of the target platform and actual program runtime t_{exec} add up to 1 s per experiment.

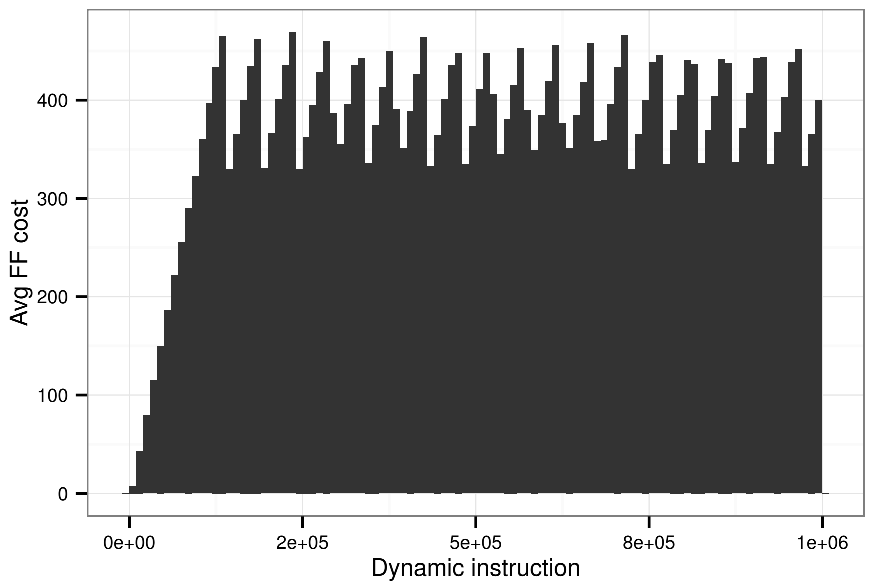


Figure 6.5: Average fast-forwarding costs for the first 10^6 dynamic instructions of the "small" BITCOUNT benchmark after adding checkpoints (cf. Figure 6.4): Fast-forwarding costs are capped at the specified threshold of 500, transforming the linearly growing costs from Figure 6.3b to a sawtooth growth curve.

it exceeds the runtime for a checkpoint load, and should be tuned to even higher values to reduce the number of checkpoints that must be created and stored.

Assuming checkpoint load costs of 300,[5] and a threshold of 500, the "small" BITCOUNT benchmark's fast-forwarding costs can be reduced to an average of 377.4 (max: 500) with only 387 checkpoints placed program-phase specifically by the heuristic. Figure 6.4 shows the distribution of these checkpoints over the benchmark's runtime: While some program phases are very impervious to smart-hopping and therefore interspersed with checkpoints – up to 11 per 200 000 dynamic instructions, – others can be fast-forwarded to without any need for additional checkpoints. Figure 6.5 shows the average cost curve from Figure 6.3b after adding checkpoints: The costs are capped at the specified threshold of 500.

6.5.2 *Alternative and Complementary Improvements*

An alternative for dealing with programs unfavorable for smart-hopping includes hardware breakpoint/watchpoint support that can be configured to trigger not at the first, but the N^{th} occurence of the watched event. For example, the Lauterbach TRACE32 debugger attached to a Hitachi SH-4 or Freescale MPC5500 microcontroller offers this feature. Unfortunately, these products are very expensive. Similarly, some CPUs come with hardware

5 Measurements on the PandaBoard indicate that it takes about 20 s, or (rounded up) ca. 300 cost units with 70 ms each, to load 2 MiB of data from the host PC into the target's memory.

performance-counting units that can be exploited to fast-forward a specified number of, for example, branch instructions. However, in many instances these counters only work "approximately accurate" [ARM11, WTM13], counteracting the task of forwarding to an exact dynamic instruction. More widely available hardware support for fast-forwarding would certainly be desirable.

Another complementary approach is to reduce the debugger round-trip times ($t_{\text{singlestep}}, t_{BP}, \ldots$), which directly improves all previously mentioned fast-forwarding techniques. For example, Heinig et al. [HKS⁺13] showed that JTAG debugging round-trip times can be reduced to around 2 ms by implementing all timing-critical operations on a microcontroller, removing all USB and host-PC related delays from the fast-forwarding operation.

6.5.3 *Def/Use Equivalence-Class Aware Fast-Forwarding*

Even more optimization potential lies in the fact that on the assessment-cycle layer, FAIL*'s FI campaigns are based on def/use equivalence classes (see Section 3.3.1.1 and Section 4.5.3.4). Instead of injecting *once* at *every dynamic instruction*, as assumed in the evaluation in Section 6.4, a def/use-pruning based campaign differs in two aspects:

- Usually, *several injections* take place at one single dynamic instruction. For example, the def/use equivalence classes Figure 3.3 lead to *eight* injections – in eight separate FI experiments – at, for example, CPU cycle #11.

- Instead of injecting at *every* dynamic instruction, injections take place on the granularity of def/use equivalence classes. For example, for each equivalence class in Figure 3.3 only one FI experiment is run. The exact injection time can be selected arbitrarily from the time range the equivalence class spans; for historic reasons, FAIL*'s *Database-Campaign*/*Database-Experiment* pair injects at the *last* dynamic instruction of each equivalence class.

The first fact – that usually several injections take place at the same dynamic instruction – is exploited by the *Database-Campaign* by simply reusing an already calculated hop chain for all subsequent injections at that CPU cycle, but this alone does not further speed up the fast-forwarding process. Instead, the implementation also exploits the degree of freedom introduced by def/use equivalence classes: Instead of always injecting at the *last* dynamic instruction – the "right margin" – of each def/use class, the algorithm picks the dynamic instruction within the class's range *with the lowest fast-forwarding cost* for the injection.

The analysis of this def/use-class aware fast-forwarding extension showed that exploiting this degree of freedom further reduces the fast-forwarding costs by up to an additional 30 percent for the DIJKSTRA, QSORT, and SUSAN benchmarks. The pathologic BITCOUNT, however, only reduces by an additional 3 percent.

6.6 SUMMARY AND CONCLUSIONS

This chapter presented the *smart-hopping* heuristic, a trace-based fast-forwarding approach for running ISA-level FI experiments on embedded platforms. It only requires basic hardware support, and can easily be ported from Fail* to other FI tools. Using instruction traces from programs in the MiBench benchmarking suite, a comparison to the prevalent simple-hopping approach demonstrated that smart-hopping improves experiment throughput by several orders of magnitude for most benchmarks.

A detailed analysis of a suboptimally performing input – the BITCOUNT benchmark – illustrated a principal limitation of smart-hopping using only breakpoints and watchpoints: Without additional hardware support, programs with long-running, homogeneous loops and little to no memory accesses cannot be efficiently fast-forwarded. Subsequently I described an extension to the heuristic that automatically inserts checkpoints based on a fast-forwarding cost threshold, effectively capping costs. The resulting checkpoint positions are aligned with program-phase specific forwarding imperviousness, and the total checkpoint count – which directly results in recording time and hard-disk space consumption – can be tuned by increasing the threshold. Beyond the checkpointing solution, I discussed further optimization potential.

7 COMPARISON OF PROGRAM SUSCEPTIBILITY TO SOFT ERRORS

"Statistics are like bikinis.
What they reveal is suggestive, but what they conceal is vital."

— Aaron Levenstein

Contents

The previous two chapters approached the problem of FI-campaign computation efforts from two different angles – reducing the total experiment count (Chapter 5), and speeding up individual experiments (Chapter 6). In contrast, this chapter is concerned with the *interpretation* of FI results *after* the campaign is completed.

As described in Chapter 3, SIHFT mechanisms need to be tested, measured and *compared* in order to assess their effectiveness in the particular use case.

In the simplest scenario, an unmodified *baseline* version of a workload – with some pre-defined input – competes with a *hardened* version of the same program, and the latter is expected to exhibit increased fault resilience. The analysis of the state of the art in hardware FI in Section 3.5 indicated that the literature predominantly conducts this kind of comparison using the *fault-coverage factor metric* (see Section 3.1.2.5).

This chapter further dissects current practices in simulation-based FI with regard to transient memory errors. Substantiated by a subset of the results from the GOP case study (see Section 4.6), I identify three common pitfalls that can skew or even completely invalidate a quantitative comparison of SIHFT-protected workloads, potentially leading to wrong conclusions for the developer. Ultimately, I propose an alternative metric that can be used for benchmark comparison.

- Described in Section 7.2, a central finding of this chapter is that special care has to be taken to *avoid distorted results when using experiment-count reduction methods*, such as def/use pruning or fault sampling.

- Even more importantly, in Section 7.3 I show that the widely used *fault-coverage metric is generally unsound for comparing different programs*. Specifically, this metric is defective for the evaluation of SIHFT mechanisms applied to a benchmark program.

- As a remedy, in Section 7.4 I construct an *objective comparison metric based on extrapolated absolute failure counts*, and introduce the mathematical foundation supporting this proposition.

As a contextual frame around these contributions, the following Section 7.1 describes the simplified machine, fault, and failure models used throughout the chapter, and Section 7.5 discusses possible generalizations and implications of the findings. Section 7.6 presents related work specifically addressing the measurement issues discussed in this chapter, and Section 7.7 summarizes the chapter.

Parts of this chapter were originally published on DSN 2015 [SBS15].

7.1 SETTING THE STAGE: MACHINE, FAULT, AND FAILURE MODEL

This section establishes the fault and machine model used throughout this chapter. Subsequently, I briefly revisit the benchmarks from the GOP case study, and describe the possible failure modes these programs can exhibit.

7.1.1 *Fault and Machine Model*

To focus on the core findings of this chapter, I use a simplistic machine model. Abstracting from CPU specifics, I assume a simple RISC CPU with classic in-order execution, without any cache levels on the way to a wait-free main memory, and with a timing of one cycle per CPU instruction. The CPU executes programs from read-only memory. Section 7.5.2 discusses possible generalizations from this simple model.

On this machine, workload runs can be carried out deterministically, just as assumed for any of the FAIL* back ends. *Deterministic* does not necessarily mean that system reactions on external events, such as asynchronous device interrupts, are out of scope, but that such events are replayed at the exact same point in time during each run.

As the basic fault model, I again – as in the GOP case study (see Section 4.6) – use uniformly distributed, independent and transient single-bit flips in main memory. Additionally, I assume the ROM, holding the program instructions, to be immune to faults. Some of the findings in this chapter may apply for other fault models, too, as discussed in Section 7.5.

7.1.2 *Example Workloads and Failure Model*

As a real-world example, – substantiating two of the three pitfalls identified in the remainder of this chapter, – I reuse the BIN_SEM3 and SYNC3 benchmarks from the GOP case study (see Section 4.6.2) in two variants:

BASELINE: The unmodified test programs.

HARDENED: The CRC variant of each benchmark, more concretely the configuration with *all* classes protected (see Figure 4.12).

Instead of using the differentiated failure model of the GOP case study, this chapter reduces the number of different experiment-outcome types to *two* for simplicity – "No Effect" and "Failure". From the experiment-outcome types described in Section 4.6, two – "No Effect" and "Detected" – can be interpreted as a benign behavior that has no visible effect from the outside. I coalesce these two result types into "No Effect", and the remaining failure modes into a subsuming "Failure" type.

7.2 EXPERIMENT-COUNT REDUCTION PITFALLS

In this section, I first briefly recapitulate the single-fault assumption and two classic experiment-count reduction techniques – sampling, and def/use pruning. By examining common practices in applying these techniques, I identify two pitfalls with potentially adverse effects on FI-result interpretation, and present means to avoid them.

7.2.1 *Improbable Independent Faults*

As already described in Section 3.1.3.2, it suffices to inject *one* fault per workload run – assuming current real-world fault rates and relatively short workload runtimes. The Poisson probability that two or more faults hit one run is negligible when compared to the probability for a single fault.

In consequence, under the assumed single-bit flip fault model, we must conduct one FI experiment for every possible CPU-cycle and memory-bit coordinate to achieve full fault-space coverage. In the simplified fault-space illustration in Figure 3.1 on page 50 with $\Delta t = 12\,cycles$ and $\Delta m = 9\,\text{bit}$, $N = 12 \cdot 9 = 108$ experiments are necessary.

7.2.2 *Experiment-Count Reduction: Fault Sampling*

Unfortunately, even for small workloads, injecting one fault for each possible CPU-cycle and memory-bit coordinate is in general infeasible. For the hypothetical workload described in Section 3.1.3.2 with a runtime of $\Delta t = 1\,\text{s}$ – corresponding to 10^9 cycles for an assumed 1 GHz CPU – and $\Delta m = 1\,\text{MiB} = 2^{23}\,\text{bit}$, for a *full* fault-space scan $w = \Delta t \cdot \Delta m \approx 8.4 \cdot 10^{15}$ FI experiments would have to be conducted. Even assuming it is possible to simulate our simple CPU in real-time, this procedure would take about 266 million CPU years.

One widespread solution to this fault-space explosion problem is *fault sampling* [AAA⁺90, PMAC95, LCMV09], as already described in detail in Section 3.1.2.5. Since the distribution of faults in the fault space is assumed uniform (see Section 7.1.1), FI experiments are picked *uniformly* from this space. Consequently, the results can be used to estimate the fault-coverage factor [BCS69]. Powell et al. define this metric *"as the probability of system recovery given that a fault exists"* [PMAC95].

The fault coverage c, or – formalizing the citation from [PMAC95] – the probability $P(\text{No Effect}|1\,\text{Fault})$ or 1-$P(\text{Failure}|1\,\text{Fault})$, can be calculated after randomly picking N (time, space) coordinates from the fault space, and running an FI experiment for each of them. In each experiment, the workload is run from the beginning until the CPU cycle for the FI – the *time* component of the randomly picked coordinate from Figure 3.1 on page 50 – has been reached. The machine is then paused, the fault gets injected by flipping the bit in memory corresponding to the *space* component of the coordinate, and the machine is resumed. As described in Section 7.1.2, then the experiment outcome is observed, turning out either as "No Effect", or as "Failure". In the latter case, the failure counter F is incremented.[1]

Repeating the formula from Section 3.1.2.5, the fault coverage c can subsequently be calculated as

$$c = 1 - P(\text{Failure}|1\,\text{Fault}) = 1 - \frac{F}{N}. \tag{7.1}$$

We will see in Section 7.3 and 7.4 that the fault-coverage metric, originally only devised for the assessment of hardware systems [BCS69], is flawed for comparing software programs. However, the sampling process itself is unproblematic, as long as a sufficiently large number of samples is taken for statistically authoritative results (see Section 3.1.2.5).

1 "No Effect" results are implicitly counted as $N - F$.

7.2.3 *Experiment-Count Reduction: Def/Use Pruning*

An alternative, widely used experiment-count reduction is *def/use pruning*, as already described in detail in Section 3.3.1.1. Based on the information from a golden-run trace, the technique partitions the fault space into def/use equivalence classes, as shown in Figure 3.3 on page 70. For those equivalence classes ending with a memory *read*, one FI experiment must be conducted – its result is representative for all fault-space coordinates contained in the equivalence class. For those classes ending with a memory *write*, a "No Effect" result can be assumed without running an experiment at all.

As a result, from the $12 \cdot 9 = 108$ experiments in the illustrative example of Figure 3.1 on page 50, only 8 remain after def/use pruning in Figure 3.3. But also in real-world examples, def/use pruning – especially when applied to faults in memory – is extremely effective. For example, the *baseline* variant of the SYNC3 benchmark (see Section 7.1.2) is reduced from a raw fault-space size of $w \approx 1.2{\times}10^9$ to merely 5985. Thus, a full fault-space scan becomes feasible even on a single machine within a reasonable time frame, and without any loss of precision regarding the result information on any point in the fault space.

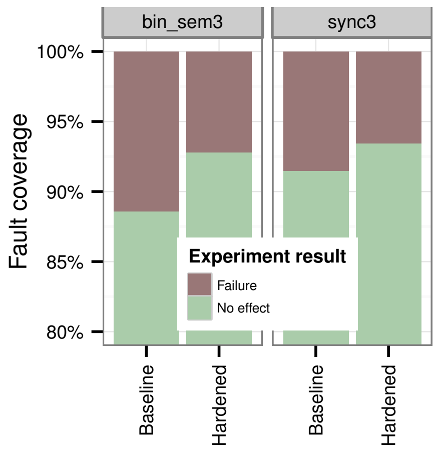
7.2.4 *Def/Use Pruning and the Weighting Pitfall*

Since its inception, def/use pruning was meant as an effort-reducing, conservative *optimization* for the theoretical full fault-space scanning model [BVFK05b, BVFK05a]. If, assuming the experiment numbers from Figure 3.3 on page 70, four of the eight (black-dotted) actually conducted experiments turned out as "Failure" – and, inversely, the remaining four as "No Effect", – this number must *not* be used in Equation 7.1 for fault-coverage calculation without any post-processing, yielding a *wrongly calculated* coverage of

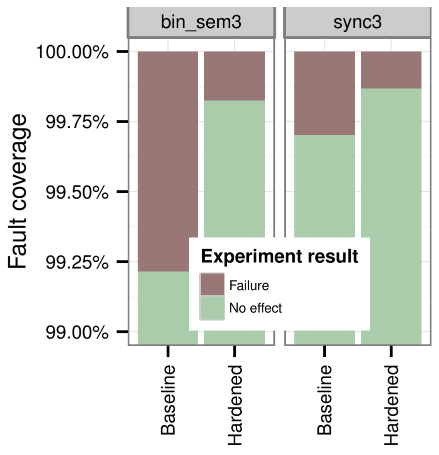
$$c = 1 - \frac{4}{8} = 50\,\%.$$

Instead, the previously collapsed equivalence classes must be expanded to their original size again, *weighting each FI-obtained result with the corresponding equivalence-class size.* Güthoff et al. also state this explicitly: *"Each result obtained by [...] fault injection experiments must be weighted with the corresponding relative data life-cycle length." [GS95]*

Going beyond their statement, the literature presents no plausible argument why fault-space coordinates we do not conduct any experiment for – the white dots in Figure 3.3, known a priori to yield "No Effect" – should be omitted in the result calculation. Section 7.3 and 7.4 will shed more light on this debate. For now, assume that *all* coordinates – the example fault-space size N is $12 \cdot 9 = 108$ – should be included in the fault-coverage calculation. The fault-coverage factor now correctly – with a weight of 7, the size of



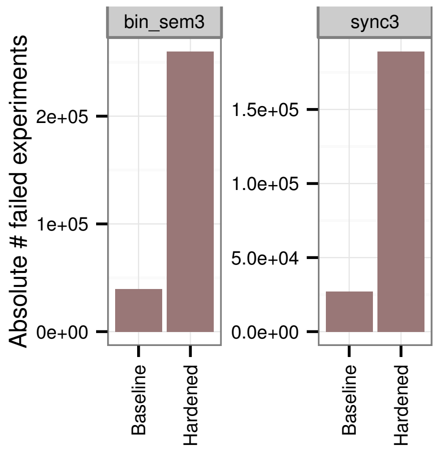
(a) Coverage, *without* result weighting.



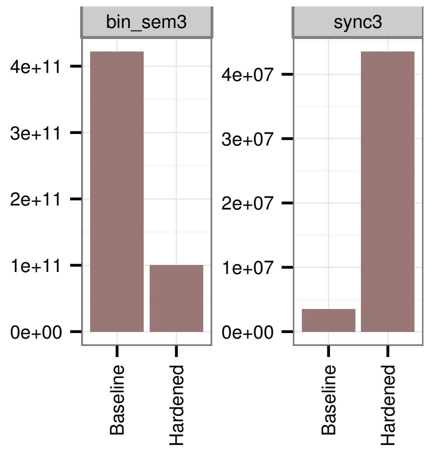
(b) Coverage, *with* result weighting.

FAULT COVERAGE	BIN_SEM3		SYNC3	
	w/o	weighted	w/o	weighted
Baseline	88.58 %	99.22 %	91.47 %	99.70 %
Hardened	92.82 %	99.83 %	93.44 %	99.87 %

(c) Fault coverage, raw data: The percentages without (w/o) weighting are off by 6.43 to 10.63 percent-points compared to the weighted coverages.



(d) Absolute failure result counts, *without* result weighting.



(e) Absolute failure result counts, *with* result weighting.

FAILURE COUNT	BIN_SEM3		SYNC3	
	w/o	weighted	w/o	weighted
Baseline	39 619	4.22×10^{11}	27 138	3.51×10^{6}
Hardened	260 053	1.01×10^{11}	189 159	4.35×10^{7}

(f) Absolute failure counts, raw data. Without (w/o) weighting, the failure counts are underestimated by several orders of magnitude.

Figure 7.1: FI-result interpretation with and without avoidance of Pitfalls 1 and 3.

each light-gray equivalence class in Figure 3.3, for each of the four "Failure" results, – calculates as

$$c = 1 - \frac{F}{N} = 1 - \frac{4 \cdot 7}{108} \approx 74.1\,\%.$$

Another explanation why this weighting is necessary can be derived from intuition: The longer data lives in a memory cell, the more probable a soft error will affect it. If no weighting is applied, the same fault coverage is calculated regardless of seven (Figure 3.3) or seven million cycles between the *store* and the subsequent *load* of the data. Then, the pruning technique is not only a methodology to reduce FI experiment efforts, but has a severe impact on the result: The *fault model* unintentionally has degenerated from "uniform transient single-bit flips in main memory" to "uniform transient single-bit flips in main memory *while a memory read operation is in progress*" – modeling something similar to single-bit flips on the memory bus, or the "inject-on-read" CPU fault model described in Section 3.1.3.3. Hence, the results are extremely skewed depending on the amount of memory accesses the benchmark executes, and the variance in memory-data lifetimes.

Now that we know def/use equivalence classes should be weighted in theory, does this have an impact on real-world examples? Figure 7.1a and 7.1b show fault coverages for the *baseline* and *hardened* variants of the BIN_SEM3 and SYNC3 benchmarks (see Section 7.1.2), calculated without and with weighting. The difference is directly visible (note the different Y-axis labels): In the unweighted case, the fault coverages of all benchmark variants are underestimated compared to the weighted case. The coverage values are off by 6.43 (SYNC3 *hardened*) up to 10.63 percent points (BIN_SEM3 *baseline*).

The reason for the bias in the two example benchmarks is a correlation between def/use equivalence class size and experiment outcome. For the four benchmark variants used in this chapter, the only positive aspect is that the *trend* from the baseline to the hardened variants is the same in both cases, so that no dangerously wrong design decisions would have been made using the unweighted coverage results. Nevertheless, the improvements achieved by the hardened program variants are not as high as Figure 7.1a suggests.

Pitfall 1: Unweighted Result Accounting

Summarizing this section, the first pitfall is the *unweighted result accounting* when using def/use pruning. Fault-space pruning is an *optimization*, and, thus, must not have any influence on the resulting numbers. When a technique such as the common def/use pruning changes the fault model's uniform distribution into a distribution that is strongly biased by the program execution, *each result must be weighted with the corresponding data lifetime* to compensate.

In the literature, a lack of result weighting is in most cases hidden behind fault-coverage factor percentages that do not reveal whether weights were applied. One example where the additionally provided data indicates that no weights were used is from Hoffmann et al. [HDL13b], who compare the fault susceptibility of two embedded operating systems using unweighted experiment numbers. Barbosa et al. [BVFK05b] recognize that weighting is needed to compensate for the effects of pruning, but conclude that the difference is small for the benchmarks they used. In contrast, our benchmark examples in Figure 7.1 serve as a warning that this is not always the case. Alexandersson and Karlsson [AK11] realize that def/use pruning affects the comparability of their results to those of other studies. Other def/use pruning descriptions simply omit the relevant detail whether weighting is used, for example Berrojo et al. [BGC+02]. Additionally, tools clearly designed for the purpose of testing – where weighting is not necessary, – such as Relyzer from Hari et al. [HANR12], may be misused for comparison purposes. Correct metrics should be integrated.

7.2.5 *Combining Def/Use Pruning and Sampling*

The conclusions from the previous section are based on def/use pruning of a *full* fault-space scan. However, often a prohibitive number of FI experiments still remain after def/use pruning. Pruning and sampling can be combined to further reduce the experiment count. Clearly, the combination of both techniques must yield the same results as pure sampling, but with reduced effort.

Therefore, samples – or, fault-space coordinates – have to be drawn uniformly from the raw, un-pruned fault space to get a representative sample of the entire fault-space population, as implemented by FAIL*'s `prune-trace` tool (see Section 4.5.3.4). The def/use pruning is then carefully applied in a second step: We only need to conduct a single FI experiment for fault-space coordinates in the sample that belong to the same def/use equivalence class. Nevertheless, we still need to count the results of *all sampled* fault-space coordinates to properly calculate the estimate for the entire fault-space population. Thereby, even coordinates known to result in "No Effect" – because their def/use equivalence class ends with a *store* (see Section 3.3.1.1), – must be included, although we will lift that requirement in Section 7.4.3.

The other way around, applying def/use pruning first and *then* drawing samples uniformly from the already-pruned fault space, – in other words, picking def/use equivalence classes with the same probability, – leads to a *biased* estimate. A fault-space coordinate that belongs to a small def/use equivalence class would be included in the sample with a higher probability than for uniform sampling of the raw fault space. The reason is that the *weight* of each equivalence class biases the selection probability of its fault-space coordinates.

Pitfall 2: Biased Sampling

Hence, the second pitfall is *biased sampling*. If def/use pruning and sampling are combined, the sampling process must *pick samples from the raw, un-pruned fault space.* If several samples belong to the same def/use equivalence class, only a single FI experiment needs to be conducted for them, but *all samples count in the estimate.*

For example, Skarin and Karlsson [SK08a] recognize that sampling from the pruned fault-space may have biased the results in their FI-based measurement of a brake-by-wire system:

> *"One factor that can contribute to the difference in the distribution of the error classification is the risk of having a biased sampling. Errors to inject were randomly selected from a list of possible errors created by the pre-injection analysis. A register or memory location that is read more often will also occur more often in this list and therefore be more likely to be selected. By adding error handling mechanisms to the most vulnerable parts of the brake controller, we also increase the number of accesses to these parts. These parts will therefore occur more often in the list of possible errors to inject. Consequently, the probability of injecting errors into the most vulnerable parts increases."* [SK08a]

However, the authors did not attempt to correct this bias.

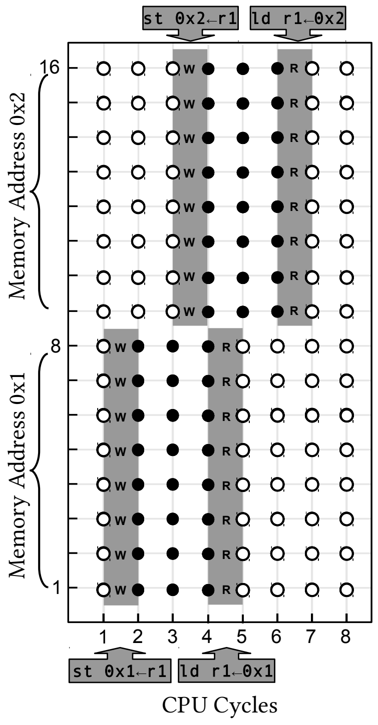
7.3 FOOLING FAULT COVERAGE: A GEDANKENEXPERIMENT

In this section, I will conduct a Gedankenexperiment with an apparently ineffective software-based hardware fault-tolerance mechanism protecting a simple benchmark program, and miraculously improving its fault coverage. Subsequently, I will revisit the fault-coverage numbers for the BIN_SEM3 and SYNC3 benchmarks, and reconsider the effectiveness of the used CRC hardening mechanism.

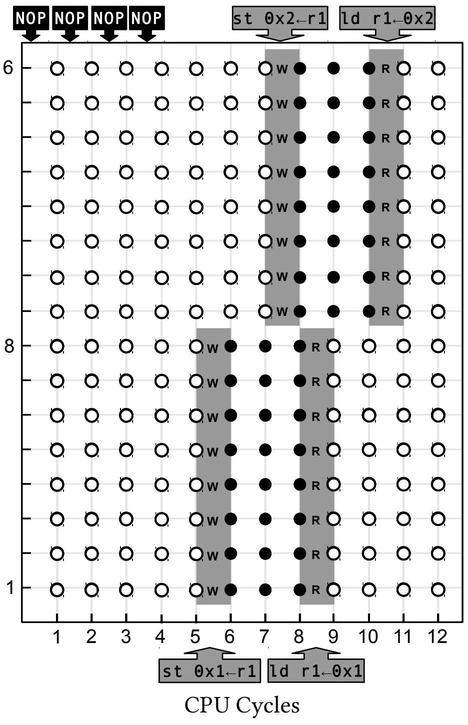
7.3.1 *"Hi", A Simple Benchmark*

Figure 7.2 shows the C-like source code for a tiny benchmark program that initializes a local character array, and subsequently communicates both character values to the outside world via the serial interface. The corresponding machine code consists of eight machine instructions consisting of four *load* and four *store* instructions. Figure 7.2a shows these loads and stores – "R" respectively "W", – similarly as in Figure 3.3, in the complete fault space of this benchmark, spanning 16 bits on the memory axis, and eight cycles on the time axis.

If we run a full fault-space scan and run one independent FI experiment for every discrete coordinate in the fault space, and observe whether the benchmark's output is identical to the golden run – it is supposed to say "Hi",



(a) Fault-space diagram for the baseline version, finishing after eight CPU cycles. Legend:
○ = effectless fault (e.g., masked),
● = fault turning into failure.



(b) Fault-space diagram after applying the *Dilution Fault Tolerance* (DFT) mechanism. Four no-operation instructions (NOP) are prepended to the baseline version, resulting in an offset of four CPU cycles.

```
volatile char msg[2];
msg[0] = 'H';
msg[1] = 'i';
serial_put_char(msg[0]);
serial_put_char(msg[1]);


/* C program */
```

```
0 ld r1 <- 'H'
1 st 0x1 <- r1
2 ld r1 <- 'i'
3 st 0x2 <- r1
4 ld r1 <- 0x1
5 st $SERIAL <- r1
6 ld r1 <- 0x2
7 st $SERIAL <- r1
```

Figure 7.2: Gedankenexperiment with C-like source code for the baseline version to the left, and corresponding machine instructions (and CPU-cycle numbers) to the right. *Adapted from [SBS15].*

– the black-dotted experiments in Figure 7.2a will turn out as a "Failure", and the other ones as "No Effect":

- In the "Failure" cases, the fault hits `msg[0]` at address `0x1` while the letter `'H'` is stored there but not yet read back – which happens in CPU instruction #5, – or analogously `msg[1]` at address `0x2` while the datum `'i'` lives there.
- The "No Effect" cases are FI experiments where the fault is subsequently overwritten – before the store cycle #1 respectively #3, – or it is not activated anymore because the program terminates – after the load cycle #4 respectively #6.

The fault coverage c_{baseline} can easily be calculated (see Section 7.2.2 and 7.2.4) by counting the number of experiments $N_{\text{baseline}} = 8 \cdot 16 = 128$ and the number of "Failure" outcomes $F_{\text{baseline}} = 3 \cdot 8 \cdot 2 = 48$, and inserting them into Equation 7.1:

$$c_{\text{baseline}} = 1 - \frac{F_{\text{baseline}}}{N_{\text{baseline}}} = 62.5\,\%$$

7.3.2 *The Fault-Space Dilution Delusion*

Now, I apply a hypothetical software-based fault-tolerance method – I call it "Dilution Fault Tolerance (DFT)", or short DFT – to the baseline's machine code by conducting a program transformation. It works by prepending four `NOP` instructions[2] to the machine code, increasing the benchmark's runtime from eight to twelve CPU cycles. Figure 7.2b shows the modified fault-space diagram for the DFT-hardened benchmark: The loads and stores have shifted four cycles to the right, and the newly added experiment dots are all "No Effect", as no live data is stored in memory before the original beginning of the benchmark.

Again calculating the fault coverage c_{hardened} with $N_{\text{hardened}} = 12 \cdot 16 = 192$ and the number of "Failure" outcomes $F_{\text{hardened}} = 3 \cdot 8 \cdot 2 = 48$ yields:

$$c_{\text{hardened}} = 1 - \frac{F_{\text{hardened}}}{N_{\text{hardened}}} = 75.0\,\%$$

Interestingly, by applying a seemingly ineffective "fault tolerance" program transformation, I increased the fault coverage by 12.5 percent points. In fact, I could arbitrarily increase the coverage to any $c_{\text{hardened}} < 100\,\%$ by inserting more `NOP`s.

Now, the attentive reader may point to the literature, and cite, for example, Barbosa et al. [BVFK05b], who argue in the context of their def/use fault-space pruning technique that never activated faults – a priori known "No Effect" results – should not be included in the coverage calculation. The newly added experiment dots in Figure 7.2b, that contributed to N_{hardened} in the above calculation would disappear again, and yield $c_{\text{hardened}} = c_{\text{baseline}}$.

2 **No op**eration, performing no real work for one cycle each.

As a reaction to such disgraceful attempts to make the DFT mechanism look bad, I would devise DFT', which replaces the `NOP`s by memory reads – for instance, alternatingly executing `ld r1 ← 0x1` and `ld r1 ← 0x2` instructions to read those two memory locations. Now, all newly added experiment dots would represent faults that actually are "activated" – by being loaded into a CPU register, and subsequently discarded. DFT' would be back at $c_{\text{hardened}} = 75.0\,\%$, this time *with* the restriction from Barbosa et al. [BVFK05b].

The central problem with this restriction is, that in a black-box technique such as FI the "activation" of a fault, turning it into an error (see Section 2.2.2), is an extremely vague business in itself. It strongly depends on the sophistication of the used fault-space pruning technique, for example:

- Not injecting into *unused* memory.
- Not injecting into state that is *known to be overwritten* afterwards.
- Not injecting into state that is *known to be masked by subsequent arithmetic operations* [HANR12].
- … – this list could be continued for a while.

In my opinion, this vague distinction between "activation" and "no activation" should therefore have no place in the context of an objective fault-tolerance benchmarking metric. Though, we will see in Section 7.4 that benchmark comparison can take place without having to decide this question at all.

To summarize, a seemingly unsuspicious – but entirely artificial – program transformation can severely skew the fault-coverage factor metric.

7.3.3 *Analyzing the Root Cause*

The remaining question is: *Why* is the fault-coverage metric unfit to discover the obvious ineffectiveness of the Dilution Fault-Tolerance mechanism – and does this metric skewing not only happen in an artificial example, but also occur in real-world examples?

The fact that I used a – hypothetical – full fault-space scan in the example, and not sampling, as most works in the community, is not the culprit: Sampling is an estimate of the whole population, and if the whole population provides a wrong answer, sampling most probably does as well. A sufficient number of samples taken from both fault spaces in Figure 7.2 would yield estimates close to the exact numbers for c_{baseline} and c_{hardened} from the previous section.

A closer look at the definition of fault coverage (Equation 7.1) suggests that *the metric itself is unfit* for benchmark *comparison*: The calculated percentages are not relative to the same base – a common N for both benchmarks – as the divisor directly depends on the benchmark's runtime, and also its memory usage.[3] Due to the usual overhead in space and time for most software-based fault-tolerance mechanisms, a different fault-space size for baseline and hardened variants must be considered the norm rather than the exception.

If a simple benchmarking cheat – the DFT is, of course, nothing more – can improve the fault coverage from 62.5 percent to 75 percent, how about the *real* fault-tolerance mechanism used on the BIN_SEM3 and SYNC3 benchmarks in Figure 7.1b? Does the CRC mechanism *really* improve the hardware fault-tolerance of these programs, or is this also a (dilution) delusion?

The next section will try and construct an objective metric that can be used for benchmark comparison, and then revisit these benchmarks again.

7.4 CONSTRUCTING AN OBJECTIVE COMPARISON METRIC

Arrived at the suspicion from the previous section that the fault-coverage factor may be unfit for the comparison of software-based fault-tolerance mechanisms, I construct an *objective comparison metric* in this section. Subsequently, I answer the question whether the CRC mechanism really improves both the BIN_SEM3 and SYNC3 benchmarks.

7.4.1 *Back to the Roots: Failure Probability*

Section 3.2.2.1 pointed out that radiation-based FI – in contrast to the (simplified) simulation-based FI I want to draw results from in this chapter – is considered *"the gold standard for assessing radiation-hardness assurance"* [QBRB13]. If we used radiation-based FI on the baseline and the hardened version of a workload, we could determine the probabilities $P(\text{Failure})_{\text{baseline}}$ and $P(\text{Failure})_{\text{hardened}}$ *under increased radiation conditions*. The probabilities include a linear factor that accounts for the increased radiation [Muk08, NIS], which cancels out in the comparison ratio quotient r,

$$r = \frac{P(\text{Failure})_{\text{hardened}}}{P(\text{Failure})_{\text{baseline}}}.$$

The hardened version improves over the baseline iff $r < 1$.

Consequently, if we want to compare benchmark variants on the basis of results from simulation-based FI, we must calculate $P(\text{Failure})$ instead of

3 The DFT could also simply have used more memory for no particular purpose instead of prolonging the workload's runtime.

$P(\text{Failure}|1\,\text{Fault})$. It can be derived by decomposing it using the law of total probability:

$$\begin{aligned} & P(\text{Failure}) \\ = \; & P(\text{Failure}|0\,\text{Faults} \vee 1\,\text{Fault} \vee 2\,\text{F.} \vee 3\,\text{F.} \vee \ldots) \\ = \; & P(\text{Failure}|0\,\text{Faults}) \cdot P(0\,\text{Faults}) + \\ & P(\text{Failure}|1\,\text{Fault}) \cdot P(1\,\text{Fault}) + \\ & P(\text{Failure}|2\,\text{Faults}) \cdot P(2\,\text{Faults}) + \\ & P(\text{Failure}|3\,\text{Faults}) \cdot P(3\,\text{Faults}) + \ldots \end{aligned}$$

$P(\text{Failure}|0\,\text{Faults})$ is known to be zero, and from Section 7.2.1 respectively Section 3.1.3.2 we know $P(\text{k Faults})$ is negligibly small for $k \geq 2$ for real-world soft-error rates and sufficiently short workload runs. Hence:

$$P(\text{Failure}) \approx P(\text{Failure}|1\,\text{Fault}) \cdot P(1\,\text{Fault}) \tag{7.2}$$

In Equation 7.2, $P(\text{Failure}|1\,\text{Fault})$ can be directly calculated from the FI results collected by a complete fault-space scan, using the number of failed experiments F, and the fault-space size w (see Section 7.2.1):

$$P(\text{Failure}|1\,\text{Fault}) = \frac{F}{w} \tag{7.3}$$

In Equation 7.2, $P(1\,\text{Fault})$ can be calculated using the Poisson probability $P_\lambda(k = 1)$ from Equation 3.1 in Section 3.1.3.2. Inserting Equation 3.1 and Equation 7.3 in Equation 7.2 yields:

$$\begin{aligned} P(\text{Failure}) &\approx \frac{F}{w} \cdot \frac{\lambda^1}{1!} e^{-\lambda} = \frac{F}{w} \cdot g \cdot w \cdot e^{-gw} \\ &= F \cdot g \cdot e^{-gw} \end{aligned} \tag{7.4}$$

g is constant for different workload runs, and may not even be exactly known, but is expected to be very small. $-gw$ is negative and depends on the fault-space size, but – using the numbers from Section 3.1.3.2 – is with $-gw \approx 1.3{\times}10^{-13}$ also so small that assuming $e^{-gw} \approx 1$ yields an error of only $1 - e^{-gw} < 10^{-12}$. Hence, the failure probability – the metric identified in the beginning of this section as the *ground truth* closest to radiation-experiment results – can be approximated to be directly proportional to the absolute number of failed experiments F:

$$P(\text{Failure}) \propto F \tag{7.5}$$

Using this proportionality, we also can calculate the comparison ratio r, knowing that $r < 1$ denotes an improvement of the *hardened* variant over the *baseline*:

$$r = \frac{P(\text{Failure})_{\text{hardened}}}{P(\text{Failure})_{\text{baseline}}} = \frac{F_{\text{hardened}}}{F_{\text{baseline}}}$$

To conclude, the number of “Failed” FI experiments from a *complete fault-space scan* is a valid metric for comparing workloads.

	BIN_SEM3		SYNC3	
	Cycles	Mem. usage	Cycles	Mem. usage
Baseline	266 610 436	25 256	5262	28 008
Hardened	266 656 644	27 192	136 927	30 232

Figure 7.3: Benchmark characteristics: Runtime in CPU cycles, and memory usage (data addresses read or written during the run) in bytes.

7.4.2 *Fault Coverage and Failure Probability in the Real World*

Figure 7.1e shows the application of this finding to the BIN_SEM3 and SYNC3 benchmarks by plotting their weighted, raw failure counts. Comparing the new results to the weighted coverage from Figure 7.1b exhibits that BIN_SEM3 indeed turns out to be protected effectively by the CRC protection scheme, the same trend predicted by the misguiding fault-coverage plot in Figure 7.1b. More surprisingly, though, SYNC3's failure probability seems to *worsen* by more than a factor of twelve compared to its baseline – a fact that was completely hidden by the fault-coverage factor, possibly resulting in a wrong design decision by the developer.

The fact that the *hardened* variant of SYNC3 has an extremely increased runtime over the *baseline*, as indicated in Figure 7.3, points at the reason: In this case, a massively increased "No Effect" rate – just as in the artificial "Hi" example in Section 7.3 – completely hid the increase in absolute "Failure" results.

Pitfall 3: Fault-Coverage Percentages for Benchmark Comparison

Subsequently, the third and most important pitfall is the *usage of fault-coverage percentages* for benchmark comparison. Unless the fault-space dimensions of two program variants are identical – which is practically never the case when effective software-based fault-tolerance mechanisms are in place, – their fault coverages are measured in percent relative to different fault-space areas, and are *by definition not comparable*. Instead, *absolute failure counts from a full fault-space scan must be used* for comparison.

Despite its unfitness for the purpose, there exists an endless amount of studies that use the fault-coverage factor as a comparison metric for program susceptibility to soft errors in memory. Examples are Fuchs's analysis of the MARS operating system [Fuc96], Rebaudengo et al. [RSRVT01] with a source-to-source compiler for dependable software, Nicolescu et al. [NSV04] analyzing a hardened space communications application, Chen et al. [CCK+06] measuring the effectiveness of object duplication in a Java runtime environment, or work I contributed to myself [BSS12] analyzing a protection scheme for virtual-function pointers in C++. Depending on the overhead the analyzed protection mechanisms introduce, – affecting the fault-space size in time or memory dimensions, – the conclusions in these studies may be worth a second look. In some cases, the evaluated mechanisms may actually worsen the system's fault resilience in the presented workload scenarios.

7.4.3 *No-Effect Results and Sampling*

For the new *absolute failure count* metric, only "Failure" results are relevant for comparison. As I demonstrated and discussed in Section 7.3, "No Effect" results can be arbitrarily skewed – either voluntarily, as in the case of the Dilution Fault Tolerance mechanism, or by chance – by artificially modifying the workload's runtime or memory usage.

Pitfall 3 (Corollary 1): "No Effect" Result Counts

Thus, Corollary 1 of Pitfall 3 is that *"No Effect" experiment outcomes are irrelevant* for the comparison of program susceptibility to soft errors in memory, and *should be excluded* from the data. Their occurrence is arbitrarily influenced by the activation and subsequent masking of faults, and ultimately by the sophistication of fault-space pruning mechanisms (see cf. Section 7.3.2).

This also means that when combining def/use pruning with sampling (see Section 7.2.5), it is not necessary to sample from equivalence classes that are known to result in "No Effect". This reduces the population size from w to $w' \leq w$.

When sampling is used (see Section 7.2.2 and 7.2.5), the number of samples N_{sampled}, and indirectly also the measured "Failure" count F_{sampled}, is arbitrarily chosen by the developer[4]. Hence, the raw F_{sampled} cannot be used directly for comparison. To get sampling results into a form usable for the new metric, the raw sample counts must be extrapolated to the fault-space size to estimate the number of "Failure" results from a full fault-space scan.

Pitfall 3 (Corollary 2): Raw Sample Counts

Thus, Corollary 2 of Pitfall 3 is *not to use raw sample counts.* If sampling is used, the raw result counts are insufficient for benchmark comparison. The *result counts must be extrapolated to the population size w* (or w', see above) to be usable for this purpose.

$$F_{\text{extrapolated}} \quad = \quad w \cdot \frac{F_{\text{sampled}}}{N_{\text{sampled}}}$$

One example from the literature that provides raw result numbers, but omits the extrapolation step from sampling to the full fault space, is from Nicolescu et al. [NSV04].

4 … but potentially also influenced by the envisaged confidence level, see Section 3.1.2.5.

Summary: Avoiding Pitfalls 1–3

To summarize, the comparison ratio r to objectively compare hardware-fault tolerant software systems must be calculated as follows:

$$r = \frac{P(\text{Failure})_{\text{hardened}}}{P(\text{Failure})_{\text{baseline}}} = \frac{w_{\text{hardened}} \cdot \frac{F_{\text{hardened,sampled}}}{N_{\text{hardened,sampled}}}}{w_{\text{baseline}} \cdot \frac{F_{\text{baseline,sampled}}}{N_{\text{baseline,sampled}}}}$$

In the case of a complete fault-space scan, w and N are equal, and the given formula reduces to

$$r = \frac{F_{\text{hardened}}}{F_{\text{baseline}}}.$$

7.5 DISCUSSION AND GENERALIZATION

In the following, I briefly revisit Pitfall 1 to get an intuition how weighting affects the absolute "Failure" counts from Pitfall 3. Subsequently, I describe possible generalizations of the findings in this chapter to other fault models. Finally, I discuss a specific case of cross-layer fault-coverage comparison, and its implications for the validity of high-level FI.

7.5.1 *Revisiting Pitfall 1: Unweighted Raw Numbers*

In Section 7.2.4, I already discussed the necessity to weight def/use equivalence classes by their corresponding data lifetimes (Pitfall 1), and demonstrated the impact of this decision on the fault coverages of the BIN_SEM3 and SYNC3 benchmarks. After being convinced that fault coverage is an inadequate metric for comparison in this context, how much influence does weighting have on the *absolute* "Failure" counts advocated in Section 7.4.2 (Pitfall 3)?

Figure 7.1d presents the absolute failure counts *without* weighting. In this case, both benchmarks seem to be *less* resilient to soft errors in their hardened variant when compared to the baseline. In contrast, the weighted results in Figure 7.1e reveal that BIN_SEM3 in fact improves – again (cf. Section 7.4.2), a wrong design decision would have been made.

This example underlines that *all three* pitfalls mentioned in this chapter must be paid attention to, as each of them independently can significantly falsify the results, and lead to incorrect design decisions.

7.5.2 *Possible Generalizations*

In Section 7.1.2, I simplified the failure modes to only "Failure" and "No Effect" types. However, the findings in this chapter can easily be generalized to more different experiment outcomes, for example, Pitfall 3 (Corollary 1) still holds: Only "No Effect" results – denoting *no* visible effect for the observer – should be excluded, while the remaining *effective* result-type counts – such as SDC or timeout – should be included in the analysis and separately extrapolated to the fault-space size (Pitfall 3, Corollary 2).

In Section 7.1.1, I strongly restricted the machine and fault model to simplify the explanations throughout this chapter. Nevertheless, some of the findings may be generalizable to both complex machines and a broader hardware fault model:

- A modern super-scalar out-of-order CPU with several cache levels would primarily change the *timing* of memory-access events. The def/use equivalence-class sizes would be derived from more detailed timing information, and therefore would gain a more accurate weight (Section 7.2.4).

- When a much more detailed simulator, for example on the flip-flop or gate level, is available, the findings may be extensible to other parts of the memory hierarchy. Every bit in the caches, the CPU registers, or the microarchitectural state of the CPU, could be part of the fault space. However, in this case different error rates g may be present within the same machine, – for example, due to different memory technologies or manufacturing processes, – requiring an adaption of the metric from Section 7.4. As an educated guess, especially Pitfall 3 may very well also be applicable in such a low-level FI scenario.

From an abstract point of view, the findings in this chapter should be applicable to any FI approach on a machine that provides timing information, operates on a set of state bits, and provides fine-grained read and write access information on this state.

7.5.3 *Cross-Layer Comparisons*

In principle, my findings may also be applicable to the results of two recent cross-layer FI studies that analyzed the general validity of high-level FI. Cho et al. [CMC$^+$13] inject into main memory and CPU registers, while Wei et al. [WTLP14] use the state of an artificial virtual-machine model for high-level FI validation. Both studies compare the results from high-level FI experiments to results from running the same workloads on fault-injected low-level simulators – simulating at the flip-flop level [CMC$^+$13], or the ISA level [WTLP14], in both cases providing a ground truth to match against.

Similarly to my findings in Section 7.3, the authors use the fault-coverage metric with different fault-space sizes. However, in their case, the different fault-space sizes are not caused by varying benchmark runtimes or memory footprints, but by the vastly different simulator models affecting both the state-space size and timing granularity.

From their analysis, Cho et al. [CMC$^+$13] conclude that high-level FI *"can result in high degrees of inaccuracies by more than an order of magnitude"*, quoting an error of up to factor 45. Without challenging the possibility that high-level FI may indeed be inaccurate, the used fault-coverage metric – with differing fault-space size quotients – may contribute significantly to this error. Consequently, the obtained result data should be reexamined using my *extrapolated absolute failure count* metric.

7.6 RELATED WORK

Section 3.1.2.5 and Section 3.1.2.6 already described classic metrics for the assessment of fault-tolerance mechanisms: The *fault-coverage factor* metric from Bouricius et al. [BCS69] defines a mathematical model that is used and instantiated by many subsequent approaches, such as the AVF [MWE$^+$03b, MWE$^+$03a] and PVF [SK08b, SK09b] metrics.[5] More recently, Rehman et al. [RSKH11] proposed the AVI, composed of values from their FVI, and recursively their IVI – which in the end is derived in a comparable way as the aforementioned AVF.

A primary reason for this widespread dependence on the fault-coverage factor metric may be that in their seminal FI paper, Arlat et al. [AAA$^+$90] initially *defined* FI to be a practical measurement method for this metric. Consequently, most FI literature sees this to be set in stone, and virtually all FI tools provide a way to measure fault coverage [CP95, HTI97, BP03].

In the literature, little work exists that scrutinizes the fault-coverage factor metric in the substantial way this chapter does:

- Generalizing the hardware-implemented fault-tolerance focused *Mean Instructions To Failure* (MITF) metric by Weaver et al. [WEMR04], Reis et al. [RCV$^+$05a] recognize the need for a metric that adequately captures the trade-off between performance and reliability of SIHFT techniques. They devise the *Mean Work To Failure* (MWTF) metric based on an application-specific definition of "work units" and FI measurements. Unlike the metric introduced in this chapter, MWTF is based on measuring the AVF implying a constant Δm. Furthermore, the authors do not derive the connection between MWTF and $P(\text{Failure})$, or the relation to common practices in the field. Interestingly, Reis himself was inconsequent in applying his findings and the MWTF metric in subsequent publications: In two papers from 2006 and 2007 [CRA06, RCA07], his group falls back to applying a metric equivalent to the fault-coverage factor, while another paper from 2006 [RCA$^+$06] presents MWTF results.

- Gawkowski et al. [GSR05] discuss SIHFT overheads in both state space and time, and propose scaling the fault-coverage factor accordingly. They discuss neither fault-space pruning nor the connection to the probability $P(\text{Failure})$. Nevertheless, their fundamental finding regarding the fault-coverage factor metric is similar to the one described in Section 7.3, so it is unfortunate that their publication [GSR05] has

5 However, both AVF and PVF weight their results by the observed data lifetimes, and, thus, avoid Pitfall 1 (see Section 7.2.4).

had little impact on the SIHFT community beyond their own research group until today.

- Vemu and Abraham [VA06] define the *method efficiency* metric, which weights the fault-coverage factor with the performance overhead of their SIHFT method. Similar to the MWTF, they as well do not factor in additional attack surface in the state-space dimension Δm, and do not provide any formal derivation of their metric.

- Demertzi et al. [DAH11] reinvent the MWTF under the name *Effective Number of Failures* (EF). It corresponds to the reciprocal value $\frac{1}{\text{MWTF}}$, and shares the same properties.

- More recently, Santini et al. [SRN⁺14] introduced a similar *Mean Workload Between Failures* (MWBF) metric parametrized with results from radiation measurements, in contrast to the simulation-based approach discussed in this chapter.

7.7 CONCLUSIONS

After a step-by-step analysis of current practices in simulation-based FI, I identified three common pitfalls in interpreting FI result data for the comparison of program susceptibility to soft errors in memory. Showing the effects on a real-world data set from the GOP case study (see Section 4.6), I demonstrated that each pitfall independently can skew or even completely invalidate the analysis, and lead to wrong conclusions regarding the effectiveness of software-based fault-tolerance. To summarize the three pitfalls:

1. Special care has to be taken when processing the FI results after def/use fault-space pruning has been applied (Pitfall 1 in Section 7.2.4).

2. Sampling combined with def/use pruning must account for different equivalence-class sizes (Pitfall 2 in Section 7.2.5).

3. The widely used *fault coverage factor* metric is inadequate for the comparison of different SIHFT-hardened program variants (Pitfall 3 in Section 7.4.2, and two corollaries in Section 7.4.3).

As a remedy, I derived an objective comparison metric that can be calculated both with full fault-space scans and from sampling results: *Absolute failure counts*, extrapolated to the fault-space size in the case of sampling. The variant of this metric avoiding all three pitfalls was presented in Section 7.4.3.

For each pitfall presented in this chapter, I found FI studies that are most probably affected. Especially the usage of the fault-coverage metric for benchmark comparison is widespread – the few examples cited in Section 7.4.2 by no means particularly stand out. Although I believe that many of the described SIHFT mechanisms would prevail, I suggest to reevaluate them with the *absolute failure count* comparison metric to sort out mechanisms that in fact *decrease* fault tolerance of programs they are deployed in.[6]

The new metric is an integral part of the result-analysis tools built into FAIL*'s assessment-cycle layer (see Section 4.5.5.1), and was used in all transient-fault case studies in Chapter 4.

6 With a similar motivation, a recent study by Shrivastava et al. [SRJW14] using the AVF metric [MWE+03b] surprisingly showed that five control-flow checking schemes – claimed effective by their original authors – actually *increase* the system vulnerability.

8 CONCLUSIONS AND OUTLOOK

"A dissertation is never finished.
You just stop writing."

— Everyone with a PhD[1]

Contents

With continuously shrinking semiconductor structure sizes and lower supply voltages, the per-device hardware-fault susceptibility is on the rise. A class of countermeasures with growing popularity is *Software-Implemented Hardware Fault Tolerance* (SIHFT), which avoids expensive hardware mechanisms and, even more importantly, can be applied application-specifically. However, SIHFT can, against intuition, cause more harm than good, because its overhead in time and space also increases the figurative "attack surface" of the system. It turns out that application-specific configuration of SIHFT is in fact a necessity rather than just an advantage.

Consequently, target programs need to be *analyzed* for particularly critical spots to harden. SIHFT-hardened programs need to be *measured and compared* throughout all development phases of the program to determine improvements or deteriorations over the baseline. Last but not least, SIHFT implementations need to be *tested.* The contributions of this dissertation focus on *Fault Injection* (FI) as a technique satisfying all these requirements – *analysis, measurement and comparison*, and *test.*

1 as quoted by Henry/Alexia Massalin in *Synthesis: An Efficient Implementation of Fundamental Operating System Services* [Mas92]

The following Section 8.1 summarizes the contributions of this dissertation, and discusses their either implementation-specific or general limitations. Section 8.2 provides an outlook with ongoing and future work, and Section 8.3 concludes the thesis.

8.1 SUMMARY

This dissertation advances the state of the art in five distinct contributions in the context of simulation- and TAP-based FI. The following sections summarize each contribution.

8.1.1 *Contribution 1: Design and Implementation of Fail**

The first contribution is the *design and implementation of an FI tool*, named Fail*, that overcomes several shortcomings in the state of the art, and enables research on the general drawbacks of simulation-based FI.

- The tool supports covering the complete ISA-level fault-space of a given workload, and enables analysis methods utilizing the detail level of the resulting data. This goal was achieved by integrating parallelization as a core feature into Fail*'s plumbing layer, and by adding fault-space pruning as an integral step to the assessment-cycle layer tooling.
- Beyond the *testing* purpose widespread in the literature, Fail* explicitly supports measurement and comparison, and provides an appropriate metric for this purpose (see contribution 4).
- By using AOP as a minimally invasive software coupling technique, Fail* attaches to existing simulator software without provoking serious maintainability issues.
- With the help of the carefully designed *Execution-Environment Abstraction* (EEA) layer, Fail* can be – transparently to the experiment designer – used to inject faults in several target back ends, using two different FI techniques. The tooling also allows to implement a wide range of different ISA-level CPU and memory fault models.

Although Fail*'s first workshop publication on its first prototype is only four years old [SHK$^+$12], the tool has already made a measurable impact on the SIHFT community. Since it had begun to be used project-internally – followed by the public GitHub release in 2014 – it was used

- in at least 14 bachelor and master theses [Unz13, Fri13, Wul13, Rad13, Taf14, Hel14, Luk14, Dem14, Die14, Sch15b, Sel15, Sch15c, Met16, Bre16],
- in at least five doctoral dissertations at three different universities [Ulb14, Döb14, Hof16, Sti16, Sch16] (including my own),
- for graduate and post-graduate lectures at FAU Erlangen and at the international *Winter School on Operating Systems* (WSOS),

- and in at least 20 publications by 21 researchers from four different research groups [BSS12, DSE13, SKSE13, BSS13a, SSE+13, HDL13b, BSS13b, HUD+14b, SRS14, HBD+14, SBS14, STE+14, HUD+14a, DL15, HLDL15, BSS15, SBS15, Stu15, DHL15, BS15].

8.1.2 *Contribution 2: Fault-Similarity Pruning*

The second contribution addresses the problem of exhaustive fault-space exploration and the accompanying huge computation efforts, which are in some cases – for example the GOP case study in Section 4.6 – measured in CPU *years*. My *Fault-Similarity Pruning* (FSP) approach, a heuristical fault-space pruning technique, tackles this problem by reducing the number of necessary FI experiments. I advance the state of the art in several aspects:

- The heuristic adapts to the injected workloads, compiler and machine specifics, and the used fault and failure models.
- It allows to freely trade the FI-experiment count for result accuracy, and allows the user to choose a Pareto-optimal heuristic configuration.
- Unlike prevalent sampling techniques, FSP provides information on *all* possible fault-space coordinates, enabling fine-grained analysis techniques (see contribution 4).

These properties allow to gain insights on local phenomena for EDM/ERM placement with massively reduced experimentation efforts: For example, if the user chooses to run 1.5 percent of all FI experiments, the average result accuracy is 99.84 percent (see Chapter 5 for more results).

8.1.3 *Contribution 3: Fast-Forwarding with Smart-Hopping*

The third contribution also addresses FI-campaign computation efforts, but from another angle: It decreases the runtime of *individual* FI experiments on real prototype hardware, controlled via a *Test-Access Port* (TAP). I identified the *fast-forwarding* operation – advancing a workload to the specific CPU cycle in its instruction stream when the fault is injected – as a major bottleneck, and advance the state of the art by a novel fast-forwarding technique named *Smart-Hopping*:

- Based on an instruction and memory-access trace, the approach uses standard debugging hardware to advance to a chosen point in program execution with a minimal number of steps.
- Compared to the state of the art, Smart-Hopping increases the speed of the fast-forwarding operation *by several orders of magnitude* for most workloads.

8.1.4 *Contribution 4: Comparison Metric*

In the fourth contribution, I dissect current practices in FI-based evaluation of SIHFT-hardened programs. I identify three pitfalls in the result interpretation, and advance the state of the art by providing a novel comparison metric:

- After uncovering another two ill-advised practices in the interpretation of FI results when using fault-space pruning or sampling, I demonstrate in a thought experiment that *the fault-coverage factor metric is unsound* for the comparison of soft-error susceptibility of programs, and can lead to wrong design decisions.
- These findings are substantiated by results from an FI campaign conducted with FAIL*, and by providing references to works where misinterpretations may have skewed the evaluation results, and possibly have influenced the drawn conclusions.
- Based on these insights, *I construct an objective metric usable for comparison* using extrapolated absolute failure counts, and introduce the mathematical foundation supporting this proposition.

The novel *extrapolated absolute failure count* metric is an integral part of the result-analysis tools built into FAIL*'s assessment-cycle layer, and helped improving all works that used it in their evaluation.

8.1.5 *Contribution 5: Fine-Grained FI-Result Analysis*

The fifth contribution exploits the newly gained possibility to analyze FI results from complete fault-space exploration, and distills useful information for SIHFT development and placement:

- Unlike current practice in the state of the art, collecting result data on the *whole* ISA-level fault space allows postponing the decision on the analysis granularity in space and time until after the FI campaign.
- The detail level of the result data allows to visually inspect the fault space, and to run analyses aiding design decisions on the level of program modules, functions, variables, source-code lines, or single machine instructions.
- Two case studies demonstrated the utility of these analyses for iteratively developing, improving, and placing respectively configuring SIHFT mechanisms.

Similarly to contribution 4, these novel FI-result analysis methods are an integral part of the result-analysis tools built into FAIL*'s assessment-cycle layer, and has since proved invaluable for its users.

8.1.6 *Limitations*

FI in general, and some of the presented approaches in particular, are subject to a set of limitations that should not remain unmentioned.

As already described in Section 3.1.2.3, FI is *generally* limited in two aspects:

- In its purpose as a *testing* methodology for SIHFT, it can be seen as a special case of program testing. Hence, it can only be used to find flaws in the SIHFT implementation, not to prove its correctness.
- When used for *analysis* or *measurement*, FI results are highly dependent on the real-world representativeness of the chosen workloads and their inputs.[2]

Consequently, FI results always – especially when they indicate "perfect" fault tolerance of the analyzed workload – must be interpreted with care.

More specifically, *simulation-based* FI – as used throughout major parts of this dissertation – is limited by the level of detail of the used machine model. The Bochs simulator I used throughout parts of this thesis uses a very simple timing model of one instruction per cycle, and assumes wait-free access to memory. The simulator was chosen specifically for its simplicity, but FAIL* now has the interfaces to attach more realistic machine models. However, some of the findings in this dissertation would not have been possible with a slow, complex, but timing-accurate simulator. A slow FI infrastructure would have precluded full fault-space scans for most workloads even with def/use pruning – and full fault-space result data provided the fundament for all of the fine-grained analyses, constituted the basis of comparison for the FSP technique, and last but not least it was pivotal for understanding prevalent FI-result interpretation pitfalls and developing the novel comparison metric. In particular, FAIL*'s now developed FI-optimization techniques are rather a *prerequisite* for switching to a more accurate but slower simulator with FAIL*.

Nevertheless, Bochs' simple machine model, including the lack of a cache hierarchy, partially relativizes the FI results obtained with it. The results from injecting memory faults are rather a pessimistic overapproximation: A contemporary cache hierarchy would mask some main-memory bit flips, for example, when a cache line is written back to main memory.

Last but not least, the single-bit flip and eight-bit burst bit-flip fault models implemented in FAIL*'s assessment-cycle layer are less realistic than the multi-bit fault models described in recent research. However, I believe the simple single- and eight-bit fault models in FAIL* are a good enough approximation for analysis, measurement, and test: In most workloads and SIHFT mechanisms, they presumably provoke reactions similar to those from more complex fault patterns – unless, of course, the four-bit correction capability of, for example, a Hamming code is being tested, – and at least stand a chance

2 I specifically excluded this aspect in the present dissertation, and assumed given, expertly chosen, and representative workloads and inputs.

to be exhaustively covered. Besides patterns in the state-space dimension, in the time dimension the single-fault assumption – one SEU per FI run – still holds even in more complex fault models.

8.2 ONGOING AND FUTURE WORK

During the work on FAIL*, the two FI optimizations, and FI-related metrics, a multitude of new research ideas have spawned. In the following, I summarize ongoing and future work in the context of this dissertation.

8.2.1 *Ongoing: Fine-grained Analysis*

HARNESSING DEBUGGING INFORMATION Beyond the mapping information from machine instructions to high-level source code lines, FAIL*'s assessment-cycle layer could harness more parts of the DWARF debugging information, which modern compilers can generate. This information would enable more detailed fine-grained analyses:[3]

- Failures within thread stacks could be mapped to local data structures – local variables, or compiler-generated structures like return addresses. This would allow to assess, for example, the RAP mechanism's effectiveness much more in detail.
- Failures in global or stack-local data structures could be aggregated *by data type*. This would strongly simplify the configuration of AOP-based SIHFT mechanisms, such as the GOP or RAP mechanisms.
- Similarly, failures in composite data structures could be mapped down to individual data members, providing an even better basis for SIHFT placement.

OPTIMIZING FAULT-DETECTOR CONFIGURATIONS A workload can be hardened with executable fault-detection assertions, such as invariant checkers. For a program hardened with n of these assertions, fine-grained analyses could try to answer the question which should remain in the final product, and which should be disabled – because they add more to the failure count than they prevent, due to their additional attack surface. Based on full fault-space FI-result data on a workload variant with all assertions enabled, result-data transformations could approximate the failure counts for all other variants. This can help finding an optimal configuration with much less efforts than conducting 2^n separate FI campaigns, which is generally infeasible for a real-world n.[4]

3 Michael Lenz currently works on this idea in his bachelor thesis.
4 First results with this approach turn out quite promising.

8.2.2 *Ongoing/Future Work: Comparison Metric*

COMPARISON OF FI RESULTS ON DIFFERENT ABSTRACTION LEVELS In their 2013 DAC paper, Cho et al. [CMC+13] claim that ISA-level FI is inaccurate by up to factor 45, even without a *trend* towards over- or underapproximation. However, they used the fault-coverage factor metric for their cross-layer comparison, with similar issues as described in Chapter 7, such as vastly differing fault-space size quotients. Consequently, a cross-layer FI-result comparison should be conducted again with the *extrapolated absolute failure-count* metric to investigate whether ISA-level FI is really inaccurate, or the fault-coverage metric led to a wrong conclusion.[5]

While the previous three ideas are concrete and currently being worked on, the remainder of this section presents research ideas for future work.

COMPARING ISA-LEVEL FI RESULTS TO RADIATION MEASUREMENTS With a similar research question, simulation-based FI results interpreted with the new metric should be compared to radiation-based FI results, *"the gold standard for assessing radiation-hardness assurance"* [QBRB13], to investigate how close the much cheaper ISA-level results get to reality. Santini et al.'s 2016 SELSE paper [SBD+16] (work I also contributed to) takes a first step in this direction by assessing the GOP mechanism [BSS13a, BSS15] and the *d*OSEK operating system [HDL13a, HLDL15] – which were both developed in a tight assessment cycle with FAIL*– with neutron radiation at the LANSCE facility, and measuring a reliability improvement of up to 91 respectively 74 percent.

APPLICATION TO OTHER FAULT MODELS The general idea behind the unfitness of the fault-coverage metric, and the resulting new comparison metric, should be applicable to other fault models as well. Further research should investigate what relevance the new metric has for ISA-level CPU fault models, for FI in gate-level or flip-flop level simulators, or even for software faults.

8.2.3 *Future Work: Fault-Injection Optimizations*

HIERARCHICAL COMBINATION OF FI EXPERIMENTS As an alternative *FI-experiment count reduction* approach, the fact that a large percentage of FI experiments results in "No Effect" could be exploited. A new campaign strategy could inject *multiple bit-flips at once.* Assuming that faults in these different locations do not extinguish each other – because, for example, a datum and its parity bit are hit at once, – a "No Effect" of such a combined experiment would allow skipping the corresponding single-bit experiments. Only if the outcome is any kind of failure, the campaign must – for example,

5 Mark Breddemann is currently finalizing his work on this idea in his master thesis [Bre16].

with binary search – investigate the bits more in detail that were part of the combined experiment.

SPEEDUP BY INSTRUCTION-SEQUENCE MODULARIZATION An idea to further reduce the *execution time for single experiments* exploits the deterministic execution of FI experiments. It is, again, based on the assumption that in many cases, after the FI a workload very quickly returns to the track lined out by the golden run. Before the FI campaign begins – and probably as well during the campaign in a successive learning process – the approach would modularize suitable instruction sequences from the golden run, determine its input and output data set, and subsequently record actual inputs and outputs for each sequence. Once during an FI experiment one of these instruction sequence starts and is provided a known input, the sequence can be completely skipped by replaying the known output, cutting short especially many of the "No Effect" experiments.

8.3 FINAL REMARKS

To conclude, this dissertation advances the state of the art with five distinct contributions. They include a FI tool avoiding several shortcomings in the existing FI-tool landscape, two separate and complementary techniques speeding up FI campaigns by up to several orders of magnitude, a novel FI-result interpretation metric usable for program comparison, and a variety of different, fine-grained FI-result analysis techniques exploiting results from complete fault-space scans. Building on these results, several research opportunities in the context of FI and SIHFT open up, and can be followed by further extending the open-source FAIL* tool set.

APPENDIX

LIST OF FIGURES

LIST OF TABLES

LISTINGS

[AAA+90] Jean Arlat, Martine Aguera, Louis Amat, Yves Crouzet, Jean-Charles Fabre, Jean-Claude Laprie, Eliane Martins, and David Powell. Fault injection for dependability validation: A methodology and some applications. *IEEE Transactions on Software Engineering*, 16(2):166–182, February 1990. doi: 10.1109/32.44380. (Cited on pages 4, 49, 62, 188, and 203.)

[ACC+93] Jean Arlat, Alain Costes, Yves Crouzet, Jean-Claude Laprie, and David Powell. Fault injection and dependability evaluation of fault-tolerant systems. *IEEE Transactions on Computers*, 42(8):913–923, August 1993. doi: 10.1109/12.238482. (Cited on page 62.)

[ACK+03] Jean Arlat, Yves Crouzet, Johan Karlsson, Peter Folkesson, Emmerich Fuchs, and Günther H. Leber. Comparison of physical and software-implemented fault injection techniques. *IEEE Transactions on Computers*, 52(9):1115–1133, September 2003. doi: 10.1109/TC.2003.1228509. (Cited on page 60.)

[ACL89] Jean Arlat, Yves Crouzet, and Jean-Claude Laprie. Fault injection for dependability validation of fault-tolerant computing systems. In *Proceedings of the 19th Annual International Symposium on Fault-Tolerant Computing (FTCS '89)*, pages 348–355, 1989. (Cited on page 62.)

[Afo09] Francisco Carlos Afonso. *Operating System Fault Tolerance Support for Real-Time Embedded Applications.* Dissertation, Universidade do Minho, Escola de Engenharia, January 2009. (Cited on page 45.)

[AFRS02] Jean Arlat, Jean-Charles Fabre, Manuel Rodríguez, and Frédéric Salles. Dependability of COTS microkernel-based systems. *IEEE Transactions on Computers*, 51:138–163, February 2002. doi: 10.1109/12.980005. (Cited on page 65.)

[AGM+71] Algirdas Avižienis, George C. Gilley, Francis P. Mathur, David A. Rennels, John A. Rohr, and David K. Rubin. The STAR (self-testing and repairing) computer: An investigation of the theory and practice of fault-tolerant computer design. *IEEE Transactions on Computers*, 20(11):1312–1321, November 1971. doi: 10.1109/T-C.1971.223133. (Cited on page 41.)

[AK11] Ruben Alexandersson and Johan Karlsson. Fault injection-based assessment of aspect-oriented implementation of fault tolerance. In *Proceedings of the 41st IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '11)*, pages 303–314, Washington, DC, USA, June 2011. IEEE Computer Society Press. doi: 10.1109/DSN.2011.5958244. (Cited on pages 45 and 192.)

[All70] Frances E. Allen. Control flow analysis. In *Proceedings of the Symposium on Compiler Optimization*, pages 1–19. ACM Press, 1970. doi: 10.1145/800028.808479. (Cited on page 42.)

[ALR01] Algirdas Avižienis, Jean-Claude Laprie, and Brian Randell. Fundamental concepts of dependability. Technical Report UCLA CSD Report 0100, Computer Science Department, University of California, Los Angeles, USA, 2001. (Cited on pages 20, 21, and 217.)

[ALR04] Algirdas Avižienis, Jean-Claude Laprie, and Brian Randell. Dependability and its threats: A taxonomy. In Renè Jacquart, editor, *Building the Information Society*, volume 156 of *IFIP International Federation for Information Processing*, pages 91–120. Springer-Verlag, 2004. doi: 10.1007/978-1-4020-8157-6_13. (Cited on pages 19, 20, and 21.)

[ALRL04] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, January 2004. doi: 10.1109/TDSC.2004.2. (Cited on pages 19, 20, 21, 22, 24, and 217.)

[Alt13] Altera Corporation. Introduction to single-event upsets. White paper, September 2013. https://www.altera.com/en_US/pdfs/literature/wp/wp-01206-introduction-single-event-upsets.pdf. (Cited on page 31.)

[AN97] Zeyad Alkhalifa and V. S. S. Nair. Design of a portable control-flow checking technique. In *Proceedings of the Workshop on High Assurance Systems Engineering*, pages 120–123. IEEE Computer Society Press, August 1997. doi: 10.1109/HASE.1997.648049. (Cited on pages 39 and 43.)

[ANKA99] Zeyad Alkhalifa, V. S. S. Nair, Narayanan Krishnamurthy, and Jacob A. Abraham. Design and evaluation of system-level checks for on-line control flow error detection. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):627–641, June 1999. doi: 10.1109/71.774911. (Cited on pages 39, 43, and 63.)

[AÖ07] Ruben Alexandersson and Peter Öhman. Implementing fault tolerance using aspect oriented programming. In *Proceedings of the 3rd Latin-American Symposium on Dependable Computing (LADC '07)*, volume 4746 of *Lecture Notes in Computer Science*, pages 57–74. Springer-Verlag, 2007. doi: 10.1007/978-3-540-75294-3_6. (Cited on page 45.)

[AÖ10] Ruben Alexandersson and Peter Öhman. On hardware resource consumption for aspect-oriented implementation of fault tolerance. In *Proceedings of the 8th European Dependable Computing Conference (EDCC '10)*, pages 61–66. IEEE Computer Society Press, April 2010. doi: 10.1109/EDCC.2010.17. (Cited on page 45.)

[AÖI05] Ruben Alexandersson, Peter Öhman, and Martin Ivarsson. Aspect oriented software implemented node level fault tolerance. In *9th IASTED International Conference on Software Engineering and Applications (SEA '05)*, Calgary, Alberta, Canada, November 2005. ACTA Press. (Cited on page 45.)

[AÖK10] Ruben Alexandersson, Peter Öhman, and Johan Karlsson. Aspect-oriented implementation of fault tolerance: An assessment of overhead. In *Proceedings of the 29th International Conference on Computer Safety, Reliability and Security (SAFECOMP '10)*, Lecture Notes in Computer Science, pages 466–479. Springer-Verlag, September 2010. doi: 10.1007/978-3-642-15651-9_34. (Cited on page 45.)

[ARH15] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. Lineage-driven fault injection. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*, pages 331–346. ACM Press, May/June 2015. doi: 10.1145/2723372.2723711. (Cited on page 68.)

[ARM11] ARM Limited. ARM architecture reference manual, ARMv7-A and ARMv7-R edition (DDI 0406C), November 2011. (Cited on page 183.)

[ASB$^+$08] Francisco Afonso, Carlos Silva, Nuno Brito, Sergio Montenegro, and Adriano Tavares. Aspect-oriented fault tolerance for real-time embedded systems. In *Proceedings of the 7th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS '08)*, pages 2:1–2:8, New York, NY, USA, 2008. ACM Press. doi: 10.1145/1404891.1404893. (Cited on page 45.)

[AVFK01] Joakim Aidemark, Jonny Vinter, Peter Folkesson, and Johan Karlsson. GOOFI: Generic object-oriented fault injection tool. In *Proceedings of the*

31st IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '01), pages 83–88, Washington, DC, USA, June/July 2001. IEEE Computer Society Press. (Cited on pages 63, 64, 78, and 79.)

[Avi71] Algirdas Avižienis. Arithmetic error codes: Cost and effectiveness studies for application in digital system design. *IEEE Transactions on Computers*, 20(11):1322–1331, November 1971. doi: 10.1109/T-C.1971.223134. (Cited on page 41.)

[Bau05a] Robert C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability*, 5(3):305–316, September 2005. doi: 10.1109/TDMR.2005.853449. (Cited on pages 30 and 34.)

[Bau05b] Robert C. Baumann. Soft errors in advanced computer systems. *IEEE Design & Test of Computers*, 22(3):258–266, May 2005. doi: 10.1109/MDT.2005.69. (Cited on pages 1 and 30.)

[BBB$^+$11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The Gem5 simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, August 2011. doi: 10.1145/2024716.2024718. (Cited on pages 67, 78, 86, and 177.)

[BCPT00] Alfredo Benso, Silvia Chiusano, Paolo Ernesto Prinetto, and Luca Tagliaferri. A C/C++ source-to-source compiler for dependable applications. In *Proceedings of the 30th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '00)*, pages 71–78. IEEE Computer Society Press, June 2000. doi: 10.1109/ICDSN.2000.857517. (Cited on pages 41 and 42.)

[BCS69] W. G. Bouricius, W. C. Carter, and P. R. Schneider. Reliability modeling techniques for self-repairing computer systems. In *Proceedings of the 24th National Conference*, ACM '69, pages 295–309, New York, NY, USA, 1969. ACM Press. doi: 10.1145/800195.805940. (Cited on pages 10, 52, 188, and 203.)

[BCSS90] James H. Barton, Edward W. Czeck, Zary Z. Segall, and Daniel P. Siewiorek. Fault injection experiments using FIAT. *IEEE Transactions on Computers*, 39(4):575–582, April 1990. doi: 10.1109/12.54853. (Cited on page 64.)

[BDC11] Alfredo Benso and Stefano Di Carlo. The art of fault injection. *Control Engineering and Applied Informatics*, 13(4):9–18, June 2011. (Cited on page 52.)

[BDCDN$^+$01a] Alfredo Benso, Stefano Di Carlo, Giorgio Di Natale, Paolo Ernesto Prinetto, and Luca Tagliaferri. Control-flow checking via regular expressions. In *Proceedings of the 10th Asian Test Symposium (ATS '01)*, pages 299–303, Los Alamitos, CA, USA, November 2001. IEEE Computer Society Press. doi: 10.1109/ATS.2001.990300. (Cited on page 43.)

[BDCDN$^+$01b] Alfredo Benso, Stefano Di Carlo, Giorgio Di Natale, Luca Tagliaferri, and Paolo Ernesto Prinetto. Validation of a software dependability tool via fault injection experiments. In *7th International On-Line Testing Workshop*, pages 3–8. IEEE Computer Society Press, 2001. doi: 10.1109/OLT.2001.937809. (Cited on page 41.)

[BDCDN$^+$03] Alfredo Benso, Stefano Di Carlo, Giorgio Di Natale, Paolo Ernesto Prinetto, and Luca Tagliaferri. Data criticality estimation in software applications. In *Proceedings of the 2003 International Test Conference (ITC '03)*, Los Alamitos, CA, USA, 2003. IEEE Computer Society Press. (Cited on pages 11 and 76.)

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, October 2003. doi: 10.1145/1165389.945462. (Cited on page 67.)

[Bel05] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the 2005 USENIX Annual Technical Conference*, pages 41–46, Berkeley, CA, USA, 2005. USENIX Association. (Cited on pages 66, 67, 78, and 86.)

[BF93] Ricky W. Butler and George B. Finelli. The infeasibility of quantifying the reliability of life-critical real-time software. *IEEE Transactions on Software Engineering*, 19(1):3–12, January 1993. doi: 10.1109/32.210303. (Cited on page 48.)

[BGC+02] L. Berrojo, I. Gonzalez, F. Corno, M. S. Reorda, G. Squillero, L. Entrena, and C. Lopez. New techniques for speeding-up fault-injection campaigns. In *Proceedings of the 2002 Conference on Design, Automation & Test in Europe (DATE '02)*, pages 847–852. IEEE Computer Society Press, 2002. doi: 10.1109/DATE.2002.998398. (Cited on pages 8, 10, 70, 73, 74, and 192.)

[BGGO03] Jan Blomgren, Bo Granbom, Thomas Granlund, and Nils Olsson. Relations between basic nuclear data and single-event upsets phenomena. *MRS Bulletin*, 28:121–125, 2 2003. doi: 10.1557/mrs2003.39. (Cited on page 60.)

[BHS+95] Robert C. Baumann, Tim Hossain, Eric Smith, Shinya Murata, and Hideki Kitagawa. Boron as a primary source of radiation in high density DRAMs. In *Symposium on VLSI Technology*, pages 81–82, June 1995. doi: 10.1109/VLSIT.1995.520868. (Cited on page 27.)

[BJS07] Shekhar Borkar, Norman P. Jouppi, and Per Stenstrom. Microprocessors in the era of terascale integration. In *Proceedings of the 2007 Conference on Design, Automation & Test in Europe (DATE '07)*, pages 237–242, San Jose, CA, USA, 2007. EDA Consortium. (Cited on pages 2 and 18.)

[BLTZ03] Stefan Bleuler, Marco Laumanns, Lothar Thiele, and Eckart Zitzler. PISA — a platform and programming language independent interface for search algorithms. In Carlos M. Fonseca, Peter J. Fleming, Eckart Zitzler, Kalyanmoy Deb, and Lothar Thiele, editors, *Proceedings of the 2nd International Conference on Evolutionary Multi-Criterion Optimization (EMO '03)*, volume 2632 of *Lecture Notes in Computer Science*, pages 494–508. Springer-Verlag, April 2003. doi: 10.1007/3-540-36970-8_35. (Cited on page 159.)

[Bor05] Shekhar Y. Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, 2005. (Cited on page 1.)

[BP03] Alfredo Benso and Paolo Ernesto Prinetto. *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation.* Frontiers in electronic testing. Kluwer Academic Publishers, Boston, Dordrecht, London, 2003. (Cited on pages 4, 49, 50, 52, 54, 70, and 203.)

[BPC98] Jérome Boué, Philippe Pétillon, and Yves Crouzet. MEFISTO-L: A VHDL-based fault injection tool for the experimental assessment of fault tolerance. In *Proceedings of the 28th Annual International Symposium on Fault-Tolerant Computing (FTCS '98)*, pages 168–173. IEEE Computer Society Press, June 1998. doi: 10.1109/FTCS.1998.689467. (Cited on page 66.)

[BPRSR98a] Alfredo Benso, Paolo Prinetto, Maurizio Rebaudengo, and Matteo Sonza Reorda. A fault injection environment for microprocessor-based boards. In *Proceedings of the 1998 International Test Conference (ITC '98)*, pages 768–773.

IEEE Computer Society Press, October 1998. doi: 10.1109/TEST.1998.743259. (Cited on pages 65 and 170.)

[BPRSR98b] Alfredo Benso, Paolo Ernesto Prinetto, Maurizio Rebaudengo, and Matteo Sonza Reorda. EXFI: A low-cost fault injection system for embedded microprocessor-based boards. *ACM Transactions on Design Automation of Electronic Systems*, 3(4):626–634, October 1998. doi: 10.1145/296333.296351. (Cited on pages 58, 65, and 170.)

[Bre16] Mark Breddemann. Gütevergleich von Prozessorfehler-Injektionsergebnissen auf unterschiedlichen Abstraktionsebenen. Master thesis, Technische Universität Dortmund, April 2016. (Cited on pages 208 and 213.)

[BRIM98] Alfredo Benso, Maurizio Rebaudengo, Leonardo Impagliazzo, and Pietro Marmo. Fault-list collapsing for fault-injection experiments. In *Proceedings of the Annual Reliability and Maintainability Symposium*, pages 383–388. IEEE Computer Society Press, January 1998. doi: 10.1109/RAMS.1998.653808. (Cited on pages 8 and 70.)

[Bro60] David T. Brown. Error detecting and correcting binary codes for arithmetic operations. *IRE Transactions on Electronic Computers*, EC-9(3):333–337, September 1960. doi: 10.1109/TEC.1960.5219855. (Cited on page 41.)

[Bro83] Rita Mae Brown. *Sudden Death*. Bantam Books, Toronto, Canada, 1st edition, May 1983. (Cited on page 149.)

[BRSR99] Alfredo Benso, Maurizio Rebaudengo, and Matteo Sonza Reorda. FlexFi: A flexible fault injection environment for microprocessor-based systems. In *Proceedings of the 18th International Conference on Computer Safety, Reliability and Security (SAFECOMP '99)*, pages 323–335, London, UK, 1999. Springer-Verlag. doi: 10.1007/3-540-48249-0_28. (Cited on pages 63, 65, 79, and 172.)

[BS15] Christoph Borchert and Olaf Spinczyk. Hardening an L4 microkernel against soft errors by aspect-oriented programming and whole-program analysis. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems (PLOS '15)*, pages 1–7, New York, NY, USA, October 2015. ACM Press. doi: 10.1145/2818302.2818304. (Cited on pages 145 and 209.)

[BSH75] D. Binder, E. C. Smith, and A. B. Holman. Satellite anomalies from galactic cosmic rays. *IEEE Transactions on Nuclear Science*, 22(6):2675–2680, December 1975. doi: 10.1109/TNS.1975.4328188. (Cited on pages 4, 26, and 27.)

[BSS12] Christoph Borchert, **Horst Schirmeier**, and Olaf Spinczyk. Protecting the dynamic dispatch in C++ by dependability aspects. In *Proceedings of the 1st GI Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES '12)*, Lecture Notes in Informatics, pages 521–535. German Society of Informatics, September 2012. (Cited on pages v, 3, 10, 45, 199, and 209.)

[BSS13a] Christoph Borchert, **Horst Schirmeier**, and Olaf Spinczyk. Generative software-based memory error detection and correction for operating system data structures. In *Proceedings of the 43rd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '13)*, Washington, DC, USA, June 2013. IEEE Computer Society Press. doi: 10.1109/DSN.2013.6575308. (Cited on pages vi, 3, 9, 14, 45, 84, 118, 128, 145, 209, and 213.)

[BSS13b] Christoph Borchert, **Horst Schirmeier**, and Olaf Spinczyk. Return-address protection in C/C++ code by dependability aspects. In *Proceedings of the 2nd GI Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES '13)*, Lecture Notes in Informatics. German Society of Informatics, September 2013. (Cited on pages vi, 3, 14, 45, 84, 127, 128, 129, 132, 134, and 209.)

[BSS15] Christoph Borchert, **Horst Schirmeier**, and Olaf Spinczyk. Generic soft-error detection and correction for concurrent data structures. *IEEE Transactions on Dependable and Secure Computing*, PP(99), 2015. Pre-print. doi: 10.1109/TDSC.2015.2427832. (Cited on pages vi, 9, 14, 45, 84, 118, 123, 145, 209, and 213.)

[BST02] Pete Broadwell, Naveen Sastry, and Jonathan Traupman. FIG: A prototype tool for online verification of recovery mechanisms. In *Workshop on Self-Healing, Adaptive and Self-Managed Systems (SHAMAN '02)*. ACM Press, 2002. (Cited on page 65.)

[BT93] Dominique Brière and Pascal Traverse. AIRBUS A320/A330/A340 electrical flight controls – a family of fault-tolerant systems. In *Proceedings of the 23rd Annual International Symposium on Fault-Tolerant Computing (FTCS '93)*, pages 616–623. IEEE Computer Society Press, June 1993. doi: 10.1109/FTCS.1993.627364. (Cited on page 38.)

[BVFK05a] Raul Barbosa, Jonny Vinter, Peter Folkesson, and Johan Karlsson. An approach to reducing the cost of fault injection. In *Proceedings of Real-Time in Sweden (RTiS '05)*, pages 129–134, August 2005. (Cited on pages 70, 71, and 189.)

[BVFK05b] Raul Barbosa, Jonny Vinter, Peter Folkesson, and Johan Karlsson. Assembly-level pre-injection analysis for improving fault injection efficiency. In *Proceedings of the 5th European Dependable Computing Conference (EDCC '05)*, volume 3463, page 246. Springer-Verlag, April 2005. (Cited on pages 58, 70, 71, 172, 189, 192, 195, and 196.)

[BWK+87] S. P. Buchner, D. Wilson, K. Kang, D. Gill, J. A. Mazer, W. D. Raburn, A. B. Campbell, and A. R. Knudson. Laser simulation of single event upsets. *IEEE Transactions on Nuclear Science*, 34(6):1227–1233, December 1987. doi: 10.1109/TNS.1987.4337457. (Cited on page 61.)

[CA78] Liming Chen and Algirdas Avižienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Proceedings of the 8th Annual International Symposium on Fault-Tolerant Computing (FTCS '78)*, pages 3–9, 1978. (Cited on page 44.)

[CBS10] CBS News / AP. Toyota "unintended acceleration" has killed 89. http://www.cbsnews.com/news/toyota-unintended-acceleration-has-killed-89/, May 2010. Retrieved 2016-04-02. (Cited on page 2.)

[CCA05] Yves Crouzet, Jacques Collet, and Jean Arlat. Mitigating soft errors to prevent a hard threat to dependable computing. In *Proceedings of the 11th International On-Line Testing Symposium (IOLTS '05)*, pages 295–298, July 2005. doi: 10.1109/IOLTS.2005.42. (Cited on page 27.)

[CCDM+09] Marcello Cinque, Domenico Cotroneo, Catello Di Martino, Stefano Russo, and Alessandro Testa. AVR-INJECT: A tool for injecting faults in wireless sensor nodes. In *Proceedings of the 23rd IEEE Parallel and Distributed Processing Symposium (PDPS '09)*, pages 1–8, May 2009. doi: 10.1109/IPDPS.2009.5160907. (Cited on pages 58 and 172.)

[CCK+06] Guilin Chen, Guangyu Chen, Mahmut Kandemir, Narayanan Vijaykrishnan, and Mary Jane Irwin. Object duplication for improving reliability. In *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, ASP-DAC '06, pages 140–145. IEEE Computer Society Press, 2006. doi: 10.1145/1118299.1118343. (Cited on pages 3, 10, and 199.)

[CG12] Sławomir Chyłek and Marcin Goliszewski. QEMU-based fault injection framework. *Studia Informatica*, 33(4), 2012. (Cited on page 67.)

[CH02] Yuhua Cheng and Chenming Hu. *MOSFET modeling & BSIM3 user's guide*. Kluwer Academic Publishers, 2002. (Cited on page 25.)

[Che10] Steve Chessin. Injecting errors for fun and profit. *ACM Queue*, 8, August 2010. (Cited on pages 4 and 83.)

[Chy14] Sławomir Chyłek. Emulation based software reliability evaluation and optimization. *Przegląd Elektrotechniczny*, 90(2):121–124, 2014. doi: 10.12915/pe.2014.02.32. (Cited on page 67.)

[CMC+13] Hyungmin Cho, Shahrzad Mirkhani, Chen-Yong Cher, Jacob A. Abraham, and Subhasish Mitra. Quantitative evaluation of soft error injection techniques for robust system design. In *Proceedings of the 50th Design Automation Conference (DAC '13)*, pages 1–10. IEEE Computer Society Press, May 2013. doi: 10.1145/2463209.2488859. (Cited on pages 55, 58, 202, and 213.)

[CMR15] Luigi Carro, Álvaro Moreira, and Paolo Rech. Mitigation of soft errors: from adding selective redundancy to changing the abstraction stack. In *Proceedings of the 45th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '15)*, June 2015. Tutorial slides. (Cited on pages 18 and 60.)

[CMS95] João Carreira, Henrique S. Madeira, and João Gabriel Silva. Xception: Software fault injection and monitoring in processor functional units. In *Proceedings of the Conference on Dependable Computing for Critical Applications (DCCA '95)*, pages 135–149, September 1995. (Cited on pages 65, 79, and 81.)

[CMS98] João Carreira, Henrique S. Madeira, and João Gabriel Silva. Xception: A technique for the experimental evaluation of dependability in modern computers. *IEEE Transactions on Software Engineering*, 24(2):125–136, February 1998. doi: 10.1109/32.666826. (Cited on pages 65, 79, and 81.)

[CMV02] Pierluigi Civera, Luca Macchiarulo, and Massimo Violante. A simplified gate-level fault model for crosstalk effects analysis. In *Proceedings of the 17th IEEE International Symposium on Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT '02)*, pages 31–39. IEEE Computer Society Press, 2002. doi: 10.1109/DFTVS.2002.1173499. (Cited on page 31.)

[Con02] Cristian Constantinescu. Impact of deep submicron technology on dependability of VLSI circuits. In *Proceedings of the 32nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '02)*, pages 205–209. IEEE Computer Society Press, June 2002. doi: 10.1109/DSN.2002.1028901. (Cited on page 1.)

[Con03] Cristian Constantinescu. Trends and challenges in VLSI circuit reliability. *IEEE Micro*, 23(4):14–19, July 2003. doi: 10.1109/MM.2003.1225959. (Cited on pages 1, 22, 27, 30, and 31.)

[Con05] Cristian Constantinescu. Neutron SER characterization of microprocessors. In *Proceedings of the 35th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '05)*, pages 754–759, June/July 2005. doi: 10.1109/DSN.2005.69. (Cited on pages 1 and 30.)

[Coo98] Richard I. Cook. How complex systems fail. https://info.aiaa.org/tac/SMG/SOSTC/Shared%20Documents/How%20Complex%20Systems%20Fail.pdf, 1998. (Cited on page 47.)

[CP95] Jeffrey A. Clark and Dhiraj K. Pradhan. Fault injection: A method for validating computer-system dependability. *IEEE Computer*, 28(6):47–56, June 1995. doi: 10.1109/2.386985. (Cited on pages 4 and 203.)

[CRA06] Jonathan Chang, George A. Reis, and David I. August. Automatic instruction-level software-only recovery. In *Proceedings of the 36th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '06)*, pages 83–92, Los Alamitos, CA, USA, 2006. IEEE Computer Society Press. doi: 10.1109/DSN.2006.15. (Cited on pages 10, 40, 43, and 203.)

[CRS99] João Carlos Cunha, Mário Zenha Rela, and João Gabriel Silva. Can software implemented fault-injection be used on real-time systems? In Jan Hlavička, Erik Maehle, and András Pataricza, editors, *Proceedings of the 3rd European Dependable Computing Conference (EDCC '99)*, pages 209–226. Springer-Verlag, 1999. doi: 10.1007/3-540-48254-7_15. (Cited on page 59.)

[CS90] Edward W. Czeck and Daniel P. Siewiorek. Effects of transient gate-level faults on program behavior. In *Proceedings of the 20th Annual International Symposium on Fault-Tolerant Computing (FTCS '90)*, pages 236–243, June 1990. doi: 10.1109/FTCS.1990.89371. (Cited on page 58.)

[dAGLKPS14] Filipe de Aguiar Geissler, Fernanda Lima Kastensmidt, and José Eduardo Pereira Souza. Soft error injection methodology based on QEMU software platform. In *Proceedings of the 15th Latin American Test Workshop (LATW '14)*. IEEE Computer Society Press, March 2014. doi: 10.1109/LATW.2014.6841910. (Cited on page 67.)

[DAH11] Melina Demertzi, Murali Annavaram, and Mary Hall. Analyzing the effects of compiler optimizations on application reliability. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization (IISWC '11)*, pages 184–193. IEEE Computer Society Press, 2011. doi: 10.1109/IISWC.2011.6114178. (Cited on page 204.)

[DBG+09] Jean-Marc Daveau, Alexandre Blampey, Gilles Gasiot, Joseph Bulone, and Philippe Roche. An industrial fault injection platform for soft-error dependability analysis and hardening of complex system-on-a-chip. In *Proceedings of the IEEE International Reliability Physics Symposium (IRPS '09)*, pages 212–220. IEEE Computer Society Press, April 2009. doi: 10.1109/IRPS.2009.5173253. (Cited on pages 6, 66, and 77.)

[DBG+11] Nathan DeBardeleben, Sean Blanchard, Qiang Guan, Ziming Zhang, and Song Fu. Experimental framework for injecting logic errors in a virtual machine to profile applications for soft error resilience. In *Proceedings of the 2011 International Conference on Parallel Processing (Euro-Par '11)*, Lecture Notes in Computer Science, pages 282–291. Springer-Verlag, September 2011. doi: 10.1007/978-3-642-29740-3_32. (Cited on pages 58 and 67.)

[DBT90] Rob Dekker, Frans Beenker, and Loek Thijssen. A realistic fault model and test algorithms for static random access memories. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(6):567–572, June 1990. doi: 10.1109/43.55188. (Cited on page 101.)

[DC07] Francis M. David and Roy H. Campbell. Building a self-healing operating system. In *Proceedings of the 3rd IEEE International Symposium on Dependable, Autonomic and Secure Computing (DASC '07)*, pages 3–10. IEEE Computer Society Press, September 2007. doi: 10.1109/DASC.2007.22. (Cited on pages 78 and 79.)

[DCCC08] Francis M. David, Ellick Chan, Jeffrey Carlyle, and Roy H. Campbell. Qinject: A virtual-machine based fault injection framework. In *Proceedings of* *the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*, 2008. (Poster Presentation). (Cited on pages 67, 78, and 79.)

[Deb01] Kalyanmoy Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, Inc., New York, NY, USA, 2001. (Cited on page 158.)

[Del97] Timothy J. Dell. A white paper on the benefits of chipkill-correct ECC for PC server main memory. Ibm white paper, 1997. (Cited on page 37.)

[Dem11] Samuel Demeulemeester. Memtest86+ – an advanced memory diagnostic tool. http://www.memtest.org/, 2011. (Cited on page 100.)

[Dem14] Tony Demann. Bewertung von aspektorientierten Fehlertoleranzmechanismen im Mikrokern-Betriebssystem Fiasco. Master thesis, Technische Universität Dortmund, 2014. (Cited on page 208.)

[DGCM⁺09] Antonio Dasilva, A. Gonzalez-Calero, José-Fernán Martínez, Lourdes López, Ana-B. García, and Vicente Hernández. Design and implementation of a Java fault injector for Exhaustif SWIFI tool. In *Proceedings of the 4th International Conference on Dependability of Computer Systems (DepCos-RELCOMEX '09)*, pages 77–83, June 2009. doi: 10.1109/DepCoS-RELCOMEX.2009.27. (Cited on page 64.)

[DGFFP13] Guiseppe Di Guglielmo, Davide Ferraretto, Franco Fummi, and Graziano Pravadelli. Efficient fault simulation through dynamic binary translation for dependability analysis of embedded software. In *Proceedings of the 18th IEEE European Test Symposium (ETS '13)*. IEEE Computer Society Press, May 2013. doi: 10.1109/ETS.2013.6569351. (Cited on page 67.)

[DH12] Björn Döbel and Hermann Härtig. Who watches the watchmen? – protecting operating system reliability mechanisms. In *Proceedings of the Workshop on Hot Topics in System Dependability (HotDep '12)*. USENIX Association, October 2012. (Cited on page 44.)

[DH14] Björn Döbel and Hermann Härtig. Can we put concurrency back into redundant multithreading? In *Proceedings of the 14th ACM International Conference on Embedded Software (EMSOFT '14)*, pages 19:1–19:10. ACM Press, October 2014. doi: 10.1145/2656045.2656050. (Cited on page 44.)

[DHE12] Björn Döbel, Hermann Härtig, and Michael Engel. Operating system support for redundant multithreading. In *Proceedings of the 12th ACM International Conference on Embedded Software (EMSOFT '12)*, pages 83–92, New York, NY, USA, 2012. ACM Press. doi: 10.1145/2380356.2380375. (Cited on page 44.)

[DHL15] Christian Dietrich, Martin Hoffmann, and Daniel Lohmann. Cross-kernel control-flow-graph analysis for event-driven real-time systems. In *Proceedings of the 2015 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '15)*, New York, NY, USA, June 2015. ACM Press. doi: 10.1145/2670529.2754963. (Cited on pages 145 and 209.)

[Die14] Christian Dietrich. Static state space exploration of an OSEK real-time operating system. Master thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, September 2014. (Cited on page 208.)

[Dij59] Edsger Wybe Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, December 1959. doi: 10.1007/BF01386390. (Cited on page 175.)

[Dij72] Edsger Wybe Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, October 1972. doi: 10.1145/355604.361591. (Cited on page 52.)

[DJM96] Scott Dawson, Farnam Jahanian, and Todd Mitton. ORCHESTRA: A probing and fault injection environment for testing protocol implementations. In *Proceedings of the IEEE International Computer Performance and Dependability Symposium (IPDS '96)*, September 1996. doi: 10.1109/IPDS.1996.540200. (Cited on page 58.)

[DJMT96] Scott Dawson, Farnam Jahanian, Todd Mitton, and Teck-Lee Tung. Testing of fault-tolerant and real-time distributed systems via protocol fault injection. In *Proceedings of the 26th Annual International Symposium on Fault-Tolerant Computing (FTCS '96)*, pages 404–414, June 1996. doi: 10.1109/FTCS.1996.534626. (Cited on page 58.)

[DL13] Domenico Di Leo. *Robustness Evaluation of Software Systems through Fault Injection*. PhD thesis, Università degli Studi di Napoli Federico II, March 2013. doi: 10.6092/UNINA/FEDOA/9257. (Cited on page 52.)

[DL15] Christian Dietrich and Daniel Lohmann. The dataref versuchung. *ACM Operating Systems Review*, pages 1–10, 2015. (Cited on page 209.)

[DLAS+12] Domenico Di Leo, Fatemeh Ayatolahi, Behrooz Sangchoolie, Johan Karlsson, and Roger Johansson. On the impact of hardware faults – an investigation of the relationship between workload inputs and failure mode distributions. In *Proceedings of the 31st International Conference on Computer Safety, Reliability and Security (SAFECOMP '12)*, pages 198–209. Springer-Verlag, September 2012. doi: 10.1007/978-3-642-33678-2_17. (Cited on page 52.)

[DM06] João A. Durães and Henrique S. Madeira. Emulation of software faults: A field data study and a practical approach. *IEEE Transactions on Software Engineering*, 32(11):849–867, November 2006. doi: 10.1109/TSE.2006.113. (Cited on pages 4 and 58.)

[DMH14] Björn Döbel, Robert Muschner, and Hermann Härtig. Resource-aware replication on heterogeneous multicores: Challenges and opportunities. In *Workshop on Resource Awareness and Adaptivity in Multi-Core Computing (RACING '14)*, May 2014. (Cited on page 44.)

[DML+07] Antonio Dasilva, José-Fernán Martínez, Lourdes López, Ana-B. García, and Luis Redondo. Exhaustif: A fault injection tool for distributed heterogeneous embedded systems. In *Proceedings of the 2007 Euro American Conference on Telematics and Information Systems (EATIS '07)*, pages 17:1–17:8, New York, NY, USA, 2007. ACM Press. doi: 10.1145/1352694.1352712. (Cited on page 64.)

[Döb14] Björn Döbel. *Operating System Support for Redundant Multithreading*. Dissertation, Technische Universität Dresden, 2014. (Cited on pages 20, 21, 23, 25, 30, 33, 44, and 208.)

[DSE13] Björn Döbel, **Horst Schirmeier**, and Michael Engel. Investigating the limitations of PVF for realistic program vulnerability assessment. In *Proceedings of the 5th HiPEAC Workshop on Design for Reliability (DFR '13)*, Berlin, Germany, January 2013. (Cited on pages v, 71, 76, and 209.)

[DW11] Anand Dixit and Alan Wood. The impact of new technology on soft error rates. In *Proceedings of the IEEE International Reliability Physics Symposium (IRPS '11)*, pages 5B.4.1–5B.4.7, April 2011. doi: 10.1109/IRPS.2011.5784522. (Cited on pages 1, 30, 56, and 60.)

[DWA10] DWARF Standards Committee. *DWARF Debugging Information Format Version 4*, June 2010. (Cited on page 117.)

[DYdS$^+$10] Marc Duranton, Sami Yehia, Bjorn de Sutter, Koen de Bosschere, Albert Cohen, Babak Falsafi, Georgi Gaydadjiev, Manolis Katevenis, Jonas Maebe, Harm Munk, Nacho Navarro, Alex Ramirez, Olivier Temam, and Mateo Valero. The HiPEAC vision. Technical report, Network of Excellence on High Performance and Embedded Architecture and Compilation, 2010. (Cited on page 1.)

[Ech90] Klaus Echtle. *Fehlertoleranzverfahren*. Springer-Verlag, 1990. doi: 10.1007/978-3-642-75765-5. (Cited on pages 20, 34, and 35.)

[ES98] Klaus Echtle and João Gabriel Silva. Fehlerinjektion – ein Mittel zur Bewertung der Maßnahmen gegen Fehler in komplexen Rechensystemen. *Informatik-Spektrum*, 21(6):328–336, 1998. doi: 10.1007/s002870050113. (Cited on page 4.)

[Eva03] Chris Evans-Pughe. They come from outer space. *IEE Review*, 49(8):30–33, September 2003. doi: 10.1049/ir:20030805. (Cited on page 27.)

[Fec09] Bernhard Fechner. *Transiente Fehler in Mikroprozessoren: Mechanismen zur Erkennung, Behebung und Tolerierung*. Dissertation, FernUniversität in Hagen, 2009. (Cited on pages 23 and 33.)

[FECA04] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit. *Aspect-oriented Software Development*. Addison-Wesley, Boston, MA, USA, 1st edition, October 2004. (Cited on page 94.)

[FF00] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced SoC (OOPSLA '00)*, October 2000. (Cited on page 94.)

[FFI06] Umberto Ferraro-Petrillo, Irene Finocchi, and Giuseppe F. Italiano. The price of resiliency: A case study on sorting with memory faults. In Yossi Azar and Thomas Erlebach, editors, *Proceedings of the 14th Annual European Symposium on Algorithms (ESA '06)*, volume 4168 of *Lecture Notes in Computer Science*, pages 768–779. Springer-Verlag, 2006. doi: 10.1007/11841036_68. (Cited on page 39.)

[FFI09] Umberto Ferraro-Petrillo, Irene Finocchi, and Giuseppe F. Italiano. The price of resiliency: A case study on sorting withÂ memory faults. *Algorithmica*, 53(4):597–620, 2009. doi: 10.1007/s00453-008-9264-1. (Cited on page 39.)

[FGAF06] André Fidalgo, Manuel Gericota, Gustavo Alves, and José Ferreira. Using NEXUS compliant debuggers for real time fault injection on microprocessors. In *Proceedings of the 19th Annual Symposium on Integrated Circuits and Systems Design (SBCCI '06)*, pages 214–219. ACM Press, August/September 2006. doi: 10.1145/1150343.1150397. (Cited on pages 7, 63, 78, 97, and 172.)

[FGI05] Irene Finocchi, Fabrizio Grandoni, and Giuseppe F. Italiano. Designing reliable algorithms in unreliable memories. In Gerth Stølting Brodal and Stefano Leonardi, editors, *Proceedings of the 13th Annual European Symposium on Algorithms (ESA '05)*, volume 3669 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005. doi: 10.1007/11561071_1. (Cited on page 39.)

[FGI07] Irene Finocchi, Fabrizio Grandoni, and Giuseppe F. Italiano. Designing reliable algorithms in unreliable memories. *Computer Science Review*, 1(2):77–87, 2007. doi: 10.1016/j.cosrev.2007.10.001. (Cited on page 39.)

[FGI09] Irene Finocchi, Fabrizio Grandoni, and Giuseppe F. Italiano. Optimal resilient sorting and searching in the presence of memory faults. *Theoretical Computer Science*, 410(44):4457–4470, 2009. doi: 10.1016/j.tcs.2009.07.026. (Cited on page 39.)

[FI04] Irene Finocchi and Giuseppe F. Italiano. Sorting and searching in the presence of memory faults (without redundancy). In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC '04)*, pages 101–110. ACM Press, 2004. doi: 10.1145/1007352.1007375. (Cited on page 39.)

[FI08] Irene Finocchi and Giuseppe F. Italiano. Sorting and searching in faulty memories. *Algorithmica*, 52(3):309–332, November 2008. doi: 10.1007/s00453-007-9088-4. (Cited on page 39.)

[For89] Philippe Forin. Vital coded microprocessor principles and application for various transit systems. In *Proceedings of the IFAC IFIP/IFORS Symposium on Control, Computers, Communications in Transportation (CCCT '89)*, pages 79–84, Oxford, UK, September 1989. Pergamon Press. (Cited on page 41.)

[Fri13] Tobias Friemel. Eine konfigurierbare Softwarearchitektur für zielplattformunabhängige Fehlerinjektion. Diploma thesis, Technische Universität Dortmund, January 2013. (Cited on page 208.)

[FSK98] Peter Folkesson, Sven Svensson, and Johan Karlsson. A comparison of simulation based and scan chain implemented fault injection. In *Proceedings of the 28th Annual International Symposium on Fault-Tolerant Computing (FTCS '98)*, pages 284–293, Los Alamitos, CA, USA, June 1998. IEEE Computer Society Press. doi: 10.1109/FTCS.1998.689479. (Cited on pages 63 and 172.)

[FSS09] Christof Fetzer, Ute Schiffel, and Martin Süßkraut. AN-encoding compiler: Building safety-critical systems with commodity hardware. In *Proceedings of the 28th International Conference on Computer Safety, Reliability and Security (SAFECOMP '09)*, pages 283–296. Springer-Verlag, September 2009. doi: 10.1007/978-3-642-04468-7_23. (Cited on page 41.)

[Fuc96] Emmerich Fuchs. An evaluation of the error detection mechanisms in MARS using software-implemented fault injection. In Andrzej Hlawiczka, João Gabriel Silva, and Luca Simoncini, editors, *Proceedings of the 2nd European Dependable Computing Conference (EDCC '96)*, pages 73–90. Springer-Verlag, 1996. doi: 10.1007/3-540-61772-8_31. (Cited on pages 10, 63, and 199.)

[GDBF14] Qiang Guan, Nathan Debardeleben, Sean Blanchard, and Song Fu. F-SEFI: A fine-grained soft error fault injection tool for profiling application vulnerability. In *Proceedings of the 28th IEEE Parallel and Distributed Processing Symposium (PDPS '14)*, pages 1245–1254. IEEE Computer Society Press, May 2014. doi: 10.1109/IPDPS.2014.128. (Cited on page 67.)

[GDJ+11] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. FATE and DESTINI: A framework for cloud recovery testing. In *Proceedings of the 8th USENIX Symposium on Network Systems Design and Implementation (NSDI '11)*, Berkeley, CA, USA, 2011. USENIX Association. (Cited on pages 6, 58, and 77.)

[GDK11] N. Gligorić, Igor Dejanović, and Srđan Krčo. Performance evaluation of compact binary XML representation for constrained devices. In *International Conference on Distributed Computing in Sensor Systems (DCOSS '11)*. IEEE Computer Society Press, June 2011. doi: 10.1109/DCOSS.2011.5982183. (Cited on page 92.)

[GEJL10] Nishant J. George, Carl R. Elks, Barry W. Johnson, and John Lach. Transient fault models and AVF estimation revisited. In *Proceedings of the 40th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '10)*, pages 477–486. IEEE Computer Society Press, June/July 2010. doi: 10.1109/DSN.2010.5544276. (Cited on page 76.)

[GGO03a] Thomas Granlund, Bo Granbom, and Nils Olsson. A comparative study between two neutron facilities regarding SEU. In *Proceedings of the 7th European Conference on Radiation and Its Effects on Components and Systems (RADECS '03)*, pages 493–497. IEEE Computer Society Press, September 2003. (Cited on page 60.)

[GGO03b] Thomas Granlund, Bo Granbom, and Nils Olsson. Soft error rate increase for new generations of SRAMs. *IEEE Transactions on Nuclear Science*, 50(6):2065–2068, December 2003. doi: 10.1109/TNS.2003.821593. (Cited on page 60.)

[GGO04] Thomas Granlund, Bo Granbom, and Nils Olsson. A comparative study between two neutron facilities regarding SEU. *IEEE Transactions on Nuclear Science*, 51(5):2922–2926, October 2004. doi: 10.1109/TNS.2004.835070. (Cited on page 60.)

[GGR$^+$04] M. S. Gordon, P. Goldhagen, K. P. Rodbell, T. H. Zabel, H. H. K. Tang, J. M. Clem, and P. Bailey. Measurement of the flux and energy spectrum of cosmic-ray induced neutrons on the ground. *IEEE Transactions on Nuclear Science*, 51(6):3427–3434, December 2004. doi: 10.1109/TNS.2004.839134. (Cited on page 29.)

[GHO$^+$07] Georg Georgakos, Peter Huber, Martin Ostermayr, Ettore Amirante, and Franz Ruckerbauer. Investigation of increased multi-bit failure rate due to neutron induced SEU in advanced embedded SRAMs. In *Proceedings of the 2007 IEEE Symposium on VLSI Circuits*, pages 80–81. IEEE Computer Society Press, June 2007. doi: 10.1109/VLSIC.2007.4342774. (Cited on page 28.)

[Giu14] Cristiano Giuffrida. *Safe and Automatic Live Update*. PhD thesis, VU Amsterdam, 2014. (Cited on pages 49, 58, and 65.)

[GKI04] Weining Gu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Error sensitivity of the Linux kernel executing on PowerPC G4 and Pentium 4 processors. In *Proceedings of the 34th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '04)*, pages 887–896. IEEE Computer Society Press, June/July 2004. doi: 10.1109/DSN.2004.1311959. (Cited on page 65.)

[GKS$^+$12] Johannes Grinschgl, Armin Krieg, Christian Steger, Reinhold Weiss, Holger Bock, and Josef Haid. Efficient fault emulation using automatic pre-injection memory access analysis. In *Proceedings of the 25th IEEE System-on-Chip Conference (SOCC '12)*, pages 277–282. IEEE Computer Society Press, 2012. doi: 10.1109/SOCC.2012.6398361. (Cited on page 70.)

[GKT89] Ulf Gunneflo, Johan Karlsson, and Jan Torin. Evaluation of error detection schemes using fault injection by heavy-ion radiation. In *Proceedings of the 19th Annual International Symposium on Fault-Tolerant Computing (FTCS '89)*, pages 340–347. IEEE Computer Society Press, June 1989. doi: 10.1109/FTCS.1989.105590. (Cited on page 60.)

[GKT13] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. EDFI: A dependable fault injection tool for dependability benchmarking experiments. In *Proceedings of the 19th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC '13)*, pages 31–40, December 2013. doi: 10.1109/PRDC.2013.12. (Cited on pages 58 and 65.)

[GŁM$^+$09] Piotr Gawkowski, Maciej Ławryńczuk, Piotr M. Marusak, Piotr Tatjewski, and Janusz Sosnowski. On improving dependability of the numerical GPC algorithm. In *European Control Conference (ECC '09)*, pages 1377–1382, August 2009. (Cited on page 40.)

[GO05] Thomas Granlund and Nils Olsson. A comparative study between proton and neutron induced SEUs in SRAMs. In *Proceedings of the 8th European Conference on Radiation and Its Effects on Components and Systems (RADECS '05)*, pages PE2–1–PE2–4, September 2005. doi: 10.1109/RADECS.2005.4365592. (Cited on page 60.)

[GO06] Thomas Granlund and Nils Olsson. SEUs induced by thermal to high-energy neutrons in SRAMs. *IEEE Transactions on Nuclear Science*, 53(6):3798–3802, December 2006. doi: 10.1109/TNS.2006.880930. (Cited on page 60.)

[Goo08] Google, Inc. *Google Protocol Buffers*, 2008. (Cited on page 92.)

[Gra85] Jim Gray. Why do computers stop and what can be done about it? Technical Report 85.7, Tandem Computers, June 1985. (Cited on page 20.)

[GRE$^+$01] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE International Workshop on Workload Characterization (WWC '01)*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society Press. doi: 10.1109/WWC.2001.15. (Cited on pages 51, 150, 161, 170, and 177.)

[GRS10] Piotr Gawkowski, Tomasz Rutkowski, and Janusz Sosnowski. Improving fault handling software techniques. In *Proceedings of the 16th International On-Line Testing Symposium (IOLTS '10)*, pages 197–199. IEEE Computer Society Press, July 2010. doi: 10.1109/IOLTS.2010.5560206. (Cited on page 44.)

[GRSRV06] Olga Goloubeva, Maurizio Rebaudengo, Matteo Sonza Reorda, and Massimo Violante. *Software-Implemented Hardware Fault Tolerance.* Springer-Verlag, New York, NY, USA, 1 edition, 2006. (Cited on pages 8, 18, 21, 40, 42, 43, 49, 50, 51, and 68.)

[GS95] Jens Güthoff and Volkmar Sieh. Combining software-implemented and simulation-based fault injection into a single fault injection method. In *Proceedings of the 25th Annual International Symposium on Fault-Tolerant Computing (FTCS '95)*, pages 196–206. IEEE Computer Society Press, June 1995. doi: 10.1109/FTCS.1995.466978. (Cited on pages 8, 70, and 189.)

[GSR05] Piotr Gawkowski, Janusz Sosnowski, and B. Radko. Analyzing the effectiveness of fault hardening procedures. In *Proceedings of the 11th International On-Line Testing Symposium (IOLTS '05)*, pages 14–19. IEEE Computer Society Press, July 2005. doi: 10.1109/IOLTS.2005.16. (Cited on pages 3 and 203.)

[HA84] Kuang-Hua Huang and Jacob A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, C-33(6):518–528, June 1984. doi: 10.1109/TC.1984.1676475. (Cited on page 39.)

[Ham50] Richard Wesley Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 29(2):147–160, April 1950. doi: 10.1002/j.1538-7305.1950.tb00463.x. (Cited on pages 36 and 37.)

[HAN12] Siva Kumar Sastry Hari, Sarita V. Adve, and Helia Naeimi. Low-cost program-level detectors for reducing silent data corruptions. In *Proceedings of the 42nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '12)*, pages 1–12, Washington, DC, USA, June 2012. IEEE Computer Society Press. (Cited on pages 3, 10, 11, and 40.)

[HANR11] Siva Kumar Sastry Hari, Sarita V. Adve, Helia Naeimi, and Pradeep Ramachandran. Relyzer: Application resiliency analyzer for transient faults. In *Proceedings of the 7th Workshop on Silicon Errors in Logic – System Effects (SELSE '11)*, 2011. (Cited on pages 67, 70, 71, 151, 152, 153, 155, and 159.)

[HANR12] Siva Kumar Sastry Hari, Sarita V. Adve, Helia Naeimi, and Pradeep Ramachandran. Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '12)*, pages 123–134, New York, NY, USA, 2012. ACM Press. doi: 10.1145/2150976.2150990. (Cited on pages 6, 8, 67, 70, 71, 77, 151, 152, 153, 155, 159, 192, and 196.)

[HANR13] Siva Kumar Sastry Hari, Sarita V. Adve, Helia Naeimi, and Pradeep Ramachandran. Relyzer: Application resiliency analyzer for transient faults. *IEEE Micro*, 33(3):58–66, May 2013. doi: 10.1109/MM.2013.30. (Cited on pages 67, 70, 71, 151, 152, 153, 155, and 159.)

[HBB$^+$11] Jörg Henkel, Lars Bauer, Joachim Becker, Oliver Bringmann, Uwe Brinkschulte, Samarjit Chakraborty, Michael Engel, Rolf Ernst, Hermann Härtig, Lars Hedrich, Andreas Herkersdorf, Rüdiger Kapitza, Daniel Lohmann, Peter Marwedel, Marco Platzner, Wolfgang Rosenstiel, Ulf Schlichtmann, Olaf Spinczyk, Mehdi Tahoori, Jürgen Teich, Norbert Wehn, and Hans-Joachim Wunderlich. Design and architectures for dependable embedded systems. In Robert P. Dick and Jan Madsen, editors, *Proceedings of the 9th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '11)*, pages 69–78. ACM Press, October 2011. doi: 10.1145/2039370.2039384. (Cited on page 2.)

[HBD$^+$13] Jörg Henkel, Lars Bauer, Nikil Dutt, Puneet Gupta, Sani Nassif, Muhammad Shafique, Mehdi Tahoori, and Norbert Wehn. Reliable on-chip systems in the nano-era: Lessons learnt and future trends. In *Proceedings of the 50th Design Automation Conference (DAC '13)*, pages 99:1–99:10. ACM Press, June 2013. doi: 10.1145/2463209.2488857. (Cited on page 56.)

[HBD$^+$14] Martin Hoffmann, Christoph Borchert, Christian Dietrich, **Horst Schirmeier**, Rüdiger Kapitza, Olaf Spinczyk, and Daniel Lohmann. Effectiveness of fault detection mechanisms in static and dynamic operating system designs. In *Proceedings of the 17th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '14)*, pages 230–237. IEEE Computer Society Press, June 2014. doi: 10.1109/ISORC.2014.26. (Cited on pages vi, 3, 145, 161, and 209.)

[HBKS14] Marcel Heing-Becker, Timo Kamph, and Sibylle Schupp. Bit-error injection for software developers. In *Proceedings of the IEEE CSMR-WCRE Software Evolution Week*, pages 434–439. IEEE Computer Society Press, February 2014. doi: 10.1109/CSMR-WCRE.2014.6747212. (Cited on page 67.)

[HDL13a] Martin Hoffmann, Christian Dietrich, and Daniel Lohmann. dOSEK: A dependable RTOS for automotive applications. In *Proceedings of the 19th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC '13)*, pages 120–121, Vancouver, British Columbia, Canada, December 2013. IEEE Computer Society Press. Fast abstract. doi: 10.1109/PRDC.2013.22. (Cited on page 213.)

[HDL13b] Martin Hoffmann, Christian Dietrich, and Daniel Lohmann. Failure by design: Influence of the RTOS interface on memory fault resilience. In *Proceedings of the 2nd GI Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES '13)*, Lecture Notes in Informatics. German Society of Informatics, September 2013. (Cited on pages 191 and 209.)

[Hel14] Richard Hellwig. Automatisierte Analyse von Fehlerinjektionsexperimenten mit Fail*. Master thesis, Technische Universität Dortmund, April 2014. (Cited on pages 117 and 208.)

[HHJ90] Robert W. Horst, Richard L. Harris, and Robert L. Jardine. Multiple instruction issue in the NonStop Cyclone processor. In *Proceedings of the 17th International Symposium on Computer Architecture (ISCA '90)*, pages 216–226, May 1990. doi: 10.1109/ISCA.1990.134528. (Cited on pages 37 and 38.)

[Hil99] Martin Hiller. Error recovery error recovery using forced validity assisted by executable assertions for error detection: An experimental evaluation. In *Proceedings of the 25th EUROMICRO Conference*, volume 2, pages 105–112. IEEE Computer Society Press, September 1999. doi: 10.1109/EURMIC.1999.794768. (Cited on page 40.)

[Hil00] Martin Hiller. Executable assertions for detecting data errors in embedded control systems. In *Proceedings of the 30th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '00)*, pages 24–33. IEEE Computer Society Press, June 2000. doi: 10.1109/ICDSN.2000.857510. (Cited on pages 3 and 40.)

[Hil02] Martin Hiller. *A Software Profiling Methodology for Design and Assessment of Dependable Software*. PhD thesis, Chalmers University of Technology, 2002. (Cited on page 64.)

[HJS01] Martin Hiller, Arshad Jhumka, and Neeraj Suri. An approach for analysing the propagation of data errors in software. In *Proceedings of the 31st IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '01)*, pages 161–172, Los Alamitos, CA, USA, June/July 2001. IEEE Computer Society Press. (Cited on pages 11, 64, and 75.)

[HJS02a] Martin Hiller, Arshad Jhumka, and Neeraj Suri. On the placement of software mechanisms for detection of data errors. In *Proceedings of the 32nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '02)*, pages 135–144. IEEE Computer Society Press, June 2002. doi: 10.1109/DSN.2002.1028894. (Cited on pages 3, 11, 64, and 75.)

[HJS02b] Martin Hiller, Arshad Jhumka, and Neeraj Suri. PROPANE: An environment for examining the propagation of errors in software. *ACM SIGSOFT*, 27(4):81–85, July 2002. doi: 10.1145/566171.566184. (Cited on pages 64 and 75.)

[HJS04] Martin Hiller, Arshad Jhumka, and Neeraj Suri. EPIC: Profiling the propagation and effect of data errors in software. *IEEE Transactions on Computers*, 53(5):512–530, May 2004. doi: 10.1109/TC.2004.1275294. (Cited on pages 11, 64, and 75.)

[HK93] Yennun Huang and Chandra Kintala. Software implemented fault tolerance: Technologies and experience. In *Proceedings of the 23rd Annual International Symposium on Fault-Tolerant Computing (FTCS '93)*. IEEE Computer Society Press, June 1993. doi: 10.1109/FTCS.1993.627302. (Cited on page 44.)

[HK12] Olof Hannius and Johan Karlsson. Impact of soft errors in a jet engine controller. In Frank Ortmeier and Peter Daniel, editors, *Proceedings of the 31st International Conference on Computer Safety, Reliability and Security (SAFECOMP '12)*, volume 7612 of *Lecture Notes in Computer Science*, pages 223–234. Springer-Verlag, Heidelberg, BW, Germany, September 2012. doi: 10.1007/978-3-642-33678-2_19. (Cited on page 172.)

[HKS+13] Andreas Heinig, Ingo Korb, Florian Schmoll, Peter Marwedel, and Michael Engel. Fast and low-cost instruction-aware fault injection. In *Proceedings of the 2nd GI Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES '13)*, Lecture Notes in Informatics. German Society of Informatics, September 2013. (Cited on page 183.)

[HL06] Rainer Hähnle and Daniel Larsson. Symbolic fault injection. Technical Report 06-17, Chalmers University of Technology, Gothenburg, Sweden, 2006. (Cited on page 68.)

[HLDL15] Martin Hoffmann, Florian Lukas, Christian Dietrich, and Daniel Lohmann. dOSEK: The design and implementation of a dependability-oriented static embedded kernel. In *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications (RTAS '15)*, pages 259–270, Los Alamitos, CA, USA, April 2015. IEEE Computer Society Press. doi: 10.1109/RTAS.2015.7108449. (Cited on pages 39, 40, 145, 209, and 213.)

[HMA+01] Scott Hareland, Jose Maiz, Mohsen Alavi, Kaizad Mistry, Steve Walsta, and Changhong Dai. Impact of CMOS process scaling and SOI on the soft error rates of logic processes. In *Proceedings of the IEEE International Symposium on VLSI Technology*, pages 73–74, June 2001. doi: 10.1109/VLSIT.2001.934953. (Cited on page 30.)

[HMR+15] Andrea Höller, Georg Macher, Tobias Rauter, Johannes Iber, and Christian Kreiner. A virtual fault injection framework for reliability-aware software development. In *International Workshop on Recent Advances in the DependabIlity AssessmeNt of Complex systEms (RADIANCE '15)*, pages 69–74. IEEE Computer Society Press, June 2015. doi: 10.1109/DSN-W.2015.16. (Cited on page 67.)

[Hof16] Martin Hoffmann. *Konstruktive Zuverlässigkeit – Eine Methodik für zuverlässige Systemsoftware auf unzuverlässiger Hardware.* Dissertation, Friedrich-Alexander-Universität Erlangen-Nürnberg, April 2016. (Cited on page 208.)

[How96] Scott Howard. A background debugging mode driver package for modular microcontrollers. *Semiconductor Application Note AN1230/D, Motorola Inc.*, 1996. (Cited on pages 7 and 63.)

[HRP15] Lena Herscheid, Daniel Richter, and Andreas Polze. Hovac: A configurable fault injection framework for benchmarking the dependability of C/C++ applications. In *Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security (QRS '15)*, pages 1–10, August 2015. doi: 10.1109/QRS.2015.12. (Cited on page 58.)

[HRS95] Seungjae Han, Harold A. Rosenberg, and Kang G. Shin. DOCTOR: An integrated software fault injection environment. In *Proceedings of the International Computer Performance and Dependability Symposium*, pages 204–213, April 1995. doi: 10.1109/IPDS.1995.395831. (Cited on pages 58 and 64.)

[HSK+14] Andrea Höller, Gerhard Schönfelder, Nermin Kajtazovic, Tobias Rauter, and Christian Kreiner. FIES: A fault injection framework for the evaluation of self-tests for COTS-based safety-critical systems. In *Proceedings of the 15th International Workshop on Microprocessor Test and Verification (MTV 2014)*, pages 105–110. IEEE Computer Society Press, December 2014. doi: 10.1109/MTV.2014.27. (Cited on page 67.)

[HSS12] Andy A. Hwang, Ioan A. Stefanovici, and Bianca Schroeder. Cosmic rays don't strike twice: Understanding the nature of DRAM errors and the implications for system design. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating*

Systems (ASPLOS '12), pages 111–122, New York, NY, USA, 2012. ACM. doi: 10.1145/2150976.2150989. (Cited on pages 2, 28, and 37.)

[HTI97] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, April 1997. doi: 10.1109/2.585157. (Cited on pages 4, 50, 51, and 203.)

[HTV07] William Hoarau, Sébastien Tixeuil, and Fabien Vauchelles. FAIL-FCI: Versatile fault injection. *Future Generation Computer Systems*, 23(7):913–919, 2007. doi: 10.1016/j.future.2007.01.005. (Cited on page 84.)

[HUD$^+$14a] Martin Hoffmann, Peter Ulbrich, Christian Dietrich, **Horst Schirmeier**, Daniel Lohmann, and Wolfgang Schröder-Preikschat. Experiences with software-based soft-error mitigation using AN codes. *Software Quality Journal*, pages 1–27, November 2014. doi: 10.1007/s11219-014-9260-4. (Cited on pages vi, 8, 42, 44, 145, and 209.)

[HUD$^+$14b] Martin Hoffmann, Peter Ulbrich, Christian Dietrich, **Horst Schirmeier**, Daniel Lohmann, and Wolfgang Schröder-Preikschat. A practitioner's guide to software-based soft-error mitigation using AN-codes. In *Proceedings of the 15th IEEE International Symposium on High Assurance Systems Engineering (HASE '14)*, pages 33–40, Miami, Florida, USA, January 2014. IEEE Computer Society Press. doi: 10.1109/HASE.2014.14. (Cited on pages vi, 8, 42, 44, 145, and 209.)

[HVAN14] Siva Kumar Sastry Hari, Radha Venkatagiri, Sarita V. Adve, and Helia Naeimi. GangES: Gang error simulation for hardware resiliency evaluation. *SIGARCH Computer Architecture News*, 42(3):61–72, June 2014. doi: 10.1145/2678373.2665685. (Cited on pages 10, 73, and 74.)

[HWS10] Kashif Hameed, Rob Williams, and Jim Smith. Separation of fault tolerance and non-functional concerns: Aspect oriented patterns and evaluation. *Journal of Software Engineering and Applications*, 3(4):303–311, 2010. doi: 10.4236/jsea.2010.34036. (Cited on page 45.)

[IEC98] IEC. *IEC 61508 – Functional safety of electrical/electronic/programmable electronic safety-related systems.* International Electrotechnical Commission, Geneva, Switzerland, December 1998. (Cited on page 2.)

[II99] IEEE-ISTO. *The Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface.* IEEE Computer Society Press, 1999. (Cited on pages 7 and 63.)

[Int08] International Roadmap Committee. International technology roadmap for semiconductors, 2008 update. http://www.itrs.net/, 2008. (Cited on page 19.)

[Int10] International Roadmap Committee. International technology roadmap for semiconductors, 2010 update. http://www.itrs.net/, 2010. (Cited on page 19.)

[Int11] International Roadmap Committee. International technology roadmap for semiconductors, 2011 edn. (executive summary). http://www.itrs.net/, 2011. (Cited on page 19.)

[Int13] International Roadmap Committee. International technology roadmap for semiconductors, 2013 edn. (executive summary). http://www.itrs.net/, 2013. (Cited on pages 1 and 19.)

[ISO11] ISO. *ISO 26262-6:2011: Road vehicles – Functional safety – Part 6: Product development at the software level.* International Organization for Standardization, Geneva, Switzerland, 2011. (Cited on page 2.)

[JAR+94] Eric Jenn, Jean Arlat, Marcus Rimén, Joakim Ohlsson, and Johan Karlsson. Fault injection into VHDL models: The MEFISTO tool. In *Proceedings of the 24th Annual International Symposium on Fault-Tolerant Computing (FTCS '94)*, pages 66–75. IEEE Computer Society Press, June 1994. doi: 10.1109/FTCS.1994.315656. (Cited on page 66.)

[JED01] JEDEC Solid State Technology Association. *JESD85: Methods for Calculating Failure Rates in Units of FITs*, July 2001. (Cited on pages 23 and 29.)

[JGS11] Pallavi Joshi, Haryadi S. Gunawi, and Koushik Sen. PREFAIL: A programmable tool for multiple-failure injection. *ACM SIGPLAN Notices*, 46(10):171–188, October 2011. doi: 10.1145/2076021.2048082. (Cited on page 84.)

[JNW08] Bruce Jacob, Spencer W. Ng, and David T. Wang. *Memory Systems: Cache, DRAM, Disk.* Morgan Kaufmann Publishers Inc., 2008. (Cited on page 26.)

[Joh08] Andréas Johansson. *Robustness Evaluation of Operating Systems.* PhD thesis, TU Darmstadt, 2008. (Cited on pages 58 and 75.)

[JS05] Andréas Johansson and Neeraj Suri. Error propagation profiling of operating systems. In *Proceedings of the 35th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '05)*, pages 86–95, June/July 2005. doi: 10.1109/DSN.2005.45. (Cited on page 75.)

[JSM07] Andréas Johansson, Neeraj Suri, and Brendan Murphy. On the selection of error model(s) for OS robustness evaluation. In *Proceedings of the 37th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '07)*, pages 502–511. IEEE Computer Society Press, June 2007. doi: 10.1109/DSN.2007.71. (Cited on page 58.)

[KDC05] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the 2005 USENIX Annual Technical Conference*, Berkeley, CA, USA, April 2005. USENIX Association. (Cited on page 170.)

[KDK+89] Hermann Kopetz, Andreas Damm, Christian Koza, Marco Mulazzani, Wolfgang Schwabl, Christoph Senft, and Ralph Zainlinger. Distributed fault-tolerant real-time systems: The Mars approach. *IEEE Micro*, 9(1):25–40, February 1989. doi: 10.1109/40.16792. (Cited on pages 60 and 63.)

[KDK+14] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA '14)*, pages 361–372, Piscataway, NJ, USA, June 2014. IEEE Computer Society Press. doi: 10.1145/2678373.2665726. (Cited on page 31.)

[KDN14] Maha Kooli and Giorgio Di Natale. A survey on simulation-based fault injection tools for complex systems. In *Proceedings of the 9th International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS '14)*, pages 1–6. IEEE Computer Society Press, May 2014. doi: 10.1109/DTIS.2014.6850649. (Cited on page 4.)

[KFA+94] Johan Karlsson, Peter Folkesson, Jean Arlat, Yves Crouzet, Günther H. Leber, and Johannes Reisinger. Evaluation of the MARS fault tolerance mechanisms using three physical fault injection techniques. In *3rd IEEE International Workshop on Integrating Error Models with Fault Injection*, pages 21–22, Washington, DC, USA, 1994. IEEE Computer Society Press. (Cited on page 60.)

[KFA+95a] Johan Karlsson, Peter Folkesson, Jean Arlat, Yves Crouzet, and Günther H. Leber. Integration and comparison of three physical fault injection techniques. In Brian Randell, Jean-Claude Laprie, Hermann Kopetz, and Bev Littlewood, editors, *Predictably Dependable Computing Systems*, ESPRIT Basic Research Series, pages 309–327. Springer Berlin Heidelberg, 1995. doi: 10.1007/978-3-642-79789-7_18. (Cited on page 60.)

[KFA+95b] Johan Karlsson, Peter Folkesson, Jean Arlat, Yves Crouzet, Günther H. Leber, and Johannes Reisinger. Application of three physical fault injection techniques to the experimental assessment of the MARS architecture. In *Proceedings of the Conference on Dependable Computing for Critical Applications (DCCA '95)*, Washington, DC, USA, 1995. IEEE Computer Society Press. (Cited on page 60.)

[KGLT91] Johan Karlsson, Ulf Gunneflo, Peter Lidén, and Jan Torin. Two fault injection techniques for test of fault handling mechanisms. In *Proceedings of the 1991 International Test Conference (ITC '91)*, October 1991. doi: 10.1109/TEST.1991.519504. (Cited on pages 60 and 62.)

[KI94] Wei-lun Kao and Ravishankar K. Iyer. DEFINE: A distributed fault injection and monitoring environment. In *Proceedings of the IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pages 252–259, June 1994. doi: 10.1109/FTPDS.1994.494497. (Cited on page 64.)

[Kil76] Jack S. Kilby. Invention of the integrated circuit. *IEEE Transactions on Electron Devices*, 23(7):648–654, July 1976. doi: 10.1109/T-ED.1976.18467. (Cited on page 1.)

[KIR+99] Zbigniew Kalbarczyk, Ravishankar K. Iyer, Gregory L. Ries, Jaqdish U. Patel, Myeong S. Lee, and Yuxiao Xiao. Hierarchical simulation approach to accurate fault modeling for system dependability evaluation. *IEEE Transactions on Software Engineering*, 25(5):619–632, September/October 1999. doi: 10.1109/32.815322. (Cited on page 58.)

[KIT93] Wei-lun Kao, Ravishankar K. Iyer, and Dong Tang. FINE: A fault injection and monitoring environment for tracing the UNIX system behavior under faults. *IEEE Transactions on Software Engineering*, 19(11):1105–1118, November 1993. doi: 10.1109/32.256857. (Cited on page 64.)

[KK07] Israel Koren and C. Mani Krishna. *Fault-Tolerant Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. (Cited on pages 23 and 34.)

[KKA93] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. EMAX: An automatic extractor of high-level error models. In *Proceedings of the 9th AIAA Computing in Aerospace Conference*, pages 1297–1306. American Institute of Aeronautics and Astronautics, October 1993. doi: 10.2514/6.1993-4698. (Cited on page 58.)

[KKA95] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. FERRARI: A flexible software-based fault and error injection system. *IEEE Transactions on Computers*, 44:248–260, 1995. (Cited on pages 63 and 65.)

[KKW+15] Naghmeh Karimi, Arun Karthik Kanuparthi, Xueyang Wang, Ozgur Sinanoglu, and Ramesh Karri. MAGIC: Malicious aging in circuits/cores. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(1):5:1–5:25, April 2015. doi: 10.1145/2724718. (Cited on page 33.)

[KLD+94] J. Karlsson, P. Liden, P. Dahlgren, R. Johansson, and U. Gunneflo. Using heavy-ion radiation to validate fault-handling mechanisms. *IEEE Micro*, 14(1):8–23, February 1994. doi: 10.1109/40.259894. (Cited on pages 60 and 61.)

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, June 1997. doi: 10.1007/BFb0053381. (Cited on pages 45, 79, 85, and 93.)

[KMJM08] Naghmeh Karimi, Michail Maniatakos, Abhijit Jas, and Yiorgos Makris. On the correlation between controller faults and instruction-level errors in modern microprocessors. In *Proceedings of the 2008 International Test Conference (ITC '08)*. IEEE Computer Society Press, October 2008. doi: 10.1109/TEST.2008.4700613. (Cited on page 58.)

[KNKA96] Ghani A. Kanawati, V. S. S. Nair, Narayanan Krishnamurthy, and Jacob A. Abraham. Evaluation of integrated system-level checks for on-line error detection. In *Proceedings of the IEEE International Computer Performance and Dependability Symposium (IPDS '96)*, pages 292–301. IEEE Computer Society Press, September 1996. doi: 10.1109/IPDS.1996.540230. (Cited on page 43.)

[Koe93] Thomas Koenig. assert(3) – Linux man page. http://linux.die.net/man/3/assert, 1993. (Cited on page 40.)

[Kon08] Kai Konrad. Implications of microcontroller software and tooling on safety-critical automotive systems. Presentation at Automotive Electronics and Electrical Systems Forum, Stuttgart. Available online: http://www.ukintpress-conferences.com/conf/08eac_conf/pdf/day_3/kaikonrad.pdf, 2008. Accessed: 2015-02-25. (Cited on page 3.)

[Kor12] Ingo Korb. A cross-layer analysis of error impact on the instruction execution in an 8-bit microprocessor. Diploma thesis, Technische Universität Dortmund, 2012. (Cited on page 58.)

[KSD+97] Philip Koopman, John Sung, Christopher Dingman, Daniel Siewiorek, and Ted Marz. Comparing operating systems using robustness benchmarks. In *Proceedings of the 16th IEEE Symposium on Reliable Distributed Systems (SRDS '97)*, pages 72–79. IEEE Computer Society Press, October 1997. doi: 10.1109/RELDIS.1997.632800. (Cited on page 65.)

[Kur05] Ray Kurzweil. *The Singularity Is Near: When Humans Transcend Biology*. Viking Press Inc., 2005. http://www.singularity.com/charts/page57.html. (Cited on page 19.)

[LA04] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Los Alamitos, CA, USA, March 2004. IEEE Computer Society Press. (Cited on pages 64 and 114.)

[LAK92] Jean-Claude Laprie, Algirdas Avižienis, and Hermann Kopetz, editors. *Dependability: Basic Concepts and Terminology*. Springer-Verlag, Secaucus, NJ, USA, 1992. (Cited on pages 20 and 21.)

[Lap85] Jean-Claude Laprie. Dependable computing and fault tolerance: Concepts and terminology. In *Proceedings of the 15th Annual International Symposium on Fault-Tolerant Computing (FTCS '85)*, pages 2–11. IEEE Computer Society Press, 1985. (Cited on pages 20 and 21.)

[Law96] Kevin P. Lawton. Bochs: A portable PC emulator for Unix/X. *Linux Journal*, 1996(29es):7, 1996. (Cited on pages 86, 87, and 93.)

[LCMV09] Régis Leveugle, A. Calvez, Paolo Maistri, and Pierre Vanhauwaert. Statistical fault injection: Quantified error and confidence. In *Proceedings of the 2009 Conference on Design, Automation & Test in Europe (DATE '09)*, pages 502–506. IEEE Computer Society Press, April 2009. doi: 10.1109/DATE.2009.5090716. (Cited on pages 53, 68, 150, and 188.)

[LCP$^+$09] Galen Lyle, Shelley Chen, Karthik Pattabiraman, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. An end-to-end approach for the automatic derivation of application-aware error detectors. In *Proceedings of the 39th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '09)*, pages 584–589, June/July 2009. doi: 10.1109/DSN.2009.5270291. (Cited on page 3.)

[LCWV13] Dong Li, Zizhong Chen, Panruo Wu, and Jeffrey S. Vetter. Rethinking algorithm-based fault tolerance with a cooperative software-hardware approach. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '13)*, pages 44:1–44:12. ACM Press, 2013. doi: 10.1145/2503210.2503226. (Cited on page 39.)

[LFW$^+$15] Qining Lu, Mostafa Farahani, Jiesheng Wei, Anna Thomas, and Karthik Pattabiraman. LLFI: An intermediate code-level fault injection tool for hardware faults. In *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS '15)*, pages 11–16. IEEE Computer Society Press, August 2015. doi: 10.1109/QRS.2015.13. (Cited on pages 64 and 81.)

[LH07a] Daniel Larsson and Rainer Hähnle. Symbolic fault injection. In Bernhard Beckert, editor, *Proceedings of the 4th International Workshop on Verification (VERIFY '07)*, volume 259 of *CEUR Workshop Proceedings*, pages 85–103. CEUR-WS.org, July 2007. (Cited on page 68.)

[LH07b] Aiguo Li and Bingrong Hong. Software implemented transient fault detection in space computer. *Aerospace Science and Technology*, 11(2):245–252, 2007. doi: 10.1016/j.ast.2006.06.006. (Cited on page 44.)

[LHSC10] Xin Li, Michael C. Huang, Kai Shen, and Lingkun Chu. A realistic evaluation of memory hardware errors and software system susceptibility. In *Proceedings of the 2010 USENIX Annual Technical Conference*, Berkeley, CA, USA, 2010. USENIX Association. (Cited on pages 27, 56, and 57.)

[LJ10] Matthew Leeke and Arshad Jhumka. Towards understanding the importance of variables in dependable software. In *Proceedings of the 8th European Dependable Computing Conference (EDCC '10)*, pages 85–94, Washington, DC, USA, 2010. IEEE Computer Society Press. doi: 10.1109/EDCC.2010.20. (Cited on page 75.)

[LJ11] Matthew Leeke and Arshad Jhumka. An automated wrapper-based approach to the design of dependable software. In *Proceedings of the 4th International Conference on Dependability (DEPEND '11)*. IARIA, 2011. (Cited on pages 11, 40, and 75.)

[LMX04] Nik Looker, Malcolm Munro, and Jie Xu. WS-FIT: A tool for dependability analysis of web services. In *Proceedings of the 28th International Conference on Computer Software and Applications Conference (COMPSAC '04)*, Los Alamitos, CA, USA, 2004. IEEE Computer Society Press. doi: 10.1109/CMPSAC.2004.1342690. (Cited on page 58.)

[LRK$^+$09] Man-Lap Li, Pradeep Ramachandran, Ulya R. Karpuzcu, Siva Kumar Sastry Hari, and Sarita V. Adve. Accurate microarchitecture-level fault modeling for studying hardware faults. In *Proceedings of the 15th IEEE International Symposium on High Performance Computer Architecture (HPCA '09)*, pages 105–116. IEEE Computer Society Press, February 2009. doi: 10.1109/HPCA.2009.4798242. (Cited on page 58.)

[LSHC07] Xin Li, Kai Shen, Michael C. Huang, and Lingkun Chu. A memory soft error measurement on production systems. In *Proceedings of the 2007 USENIX Annual Technical Conference*, Berkeley, CA, USA, 2007. USENIX Association. (Cited on pages 27 and 57.)

[LT13] Jianli Li and Qingping Tan. SmartInjector: Exploiting intelligent fault injection for SDC rate analysis. In *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT 2013)*, pages 236–242. IEEE Computer Society Press, October 2013. doi: 10.1109/DFT.2013.6653612. (Cited on pages 6, 10, 58, 67, 71, 74, 77, 151, and 153.)

[Luk14] Florian Lukas. Design and implementation of a soft-error resilient OSEK real-time operating system. Master thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, May 2014. (Cited on page 208.)

[LWW+02] Michael N. Lovellette, K. S. Wood, D. L. Wood, Jim H. Beall, Philip P. Shirvani, Namsuk Oh, and Edward J. McCluskey. Strategies for fault-tolerant, space-based computing: Lessons learned from the ARGOS testbed. In *Proceedings of the 2002 IEEE Aerospace Conference*, pages 5–2109–5–2119. IEEE Computer Society Press, 2002. doi: 10.1109/AERO.2002.1035377. (Cited on pages 2, 41, and 48.)

[LX07] Nik Looker and Jie Xu. Dependability assessment of grid middleware. In *Proceedings of the 37th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '07)*, pages 125–130, June 2007. doi: 10.1109/DSN.2007.31. (Cited on page 58.)

[MAM84] Aamer Mahmood, Dorothy M. Andrews, and Edward J. McCluskey. Executable assertions and flight software. In *Proceedings of the 6th AIAA/IEEE Digital Avionics Systems Conference (DASC '84)*, pages 346–351, 1984. doi: 10.2514/6.1984-2726. (Cited on page 40.)

[Mar11] Peter Marwedel. *Embedded System Design*. Springer-Verlag, 2nd edition, 2011. doi: 10.1007/978-94-007-0257-8. (Cited on pages 22, 23, and 48.)

[Mas92] Henry Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. Dissertation, Columbia University, New York, NY, USA, 1992. UMI Order No. GAX92-32050. (Cited on page 207.)

[Mas02] Anthony Massa. *Embedded Software Development with eCos*. Prentice Hall Professional Technical Reference, 2002. (Cited on page 118.)

[MBC10] Paul D. Marinescu, Radu Banabic, and George Candea. An extensible technique for high-precision testing of recovery code. In *Proceedings of the 2010 USENIX Annual Technical Conference*, pages 23–23. USENIX Association, 2010. (Cited on page 65.)

[MC09] Paul D. Marinescu and George Candea. LFI: A practical and general library-level fault injector. In *Proceedings of the 39th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '09)*, pages 379–388, June/July 2009. doi: 10.1109/DSN.2009.5270313. (Cited on page 65.)

[McP74] W. S. McPhee. Operating system integrity in OS/VS2. *IBM Systems Journal*, 13(3):230–252, September 1974. doi: 10.1147/sj.133.0230. (Cited on page 135.)

[MCR+12] Antonio Martínez-Álvarez, Sergio A. Cuenca-Asensi, Felipe Restrepo-Calle, Francesco R. Palomo Pinto, Hipólito Guzmán-Miranda, and Miguel A. Aguirre. Compiler-directed soft error mitigation for embedded systems. *IEEE Transactions on Dependable and Secure Computing*, 9(2):159–172, March 2012. doi: 10.1109/TDSC.2011.54. (Cited on page 3.)

[MCS95] Henrique S. Madeira, João Carreira, and João Gabriel Silva. Injection of faults in complex computers. In *IEEE Workshop on Evaluation Techniques for Dependable Systems*, 1995. (Cited on pages 65, 79, and 81.)

[MCV00] Henrique S. Madeira, Diamantino Costa, and Marco Vieira. On the emulation of software faults by software fault injection. In *Proceedings of the 30th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '00)*, pages 417–426, June 2000. doi: 10.1109/ICDSN.2000.857571. (Cited on pages 4 and 58.)

[MEFR04] Shubhendu S. Mukherjee, Joel Emer, Tryggve Fossum, and Steven K. Reinhardt. Cache scrubbing in microprocessors: Myth or necessity? In *Proceedings of the 10th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC '04)*, pages 37–42. IEEE Computer Society Press, March 2004. doi: 10.1109/PRDC.2004.1276550. (Cited on page 23.)

[Mem13] Memory Alpha Star Trek Wiki. Brak'lul. http://en.memory-alpha.wikia.com/wiki/Brak'lul, 2013. (Cited on page 34.)

[MER05] Shubhendu S. Mukherjee, Joel Emer, and Steven K. Reinhardt. The soft error problem: An architectural perspective. In *Proceedings of the 11th IEEE International Symposium on High-Performance Computer Architecture (HPCA '05)*, pages 243–247. IEEE Computer Society Press, 2005. doi: 10.1109/HPCA.2005.37. (Cited on pages 23 and 31.)

[Met16] Christian Metz. Automated application of fault tolerance measures in the KESO multi-JVM. Master thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, February 2016. (Cited on page 208.)

[MFS98] Wilfrido A. Moreno, Fernando J. Falquez, and Nitin Saini. Fault tolerant design validation through laser fault injection. In *Proceedings of the 1998 2nd IEEE International Caracas Conference on Devices, Circuits and Systems*, pages 132–137. IEEE Computer Society Press, March 1998. doi: 10.1109/ICCDCS.1998.705820. (Cited on page 61.)

[MGM$^+$99] R. J. Martínez, Pedro J. Gil, G. Martín, C. Pérez, and Juan José Serrano. Experimental validation of high-speed fault-tolerant systems using physical fault injection. In *Proceedings of the Conference on Dependable Computing for Critical Applications (DCCA '99)*, pages 249–265, November 1999. doi: 10.1109/DCFTS.1999.814299. (Cited on page 62.)

[MHB$^+$05] Ricardo Jorge Maia, Luis Henriques, Ricardo Barbosa, Diamantino Costa, and Henrique S. Madeira. Xception fault injection and robustness testing framework: a case-study of testing RTEMS. In *VI Test and Fault Tolerance Workshop*, 2005. (Cited on page 63.)

[MHCM02] Ricardo Jorge Maia, Luis Henriques, Diamantino Costa, and Henrique S. Madeira. Xception – enhanced automated fault-injection environment. In *Proceedings of the 32nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '02)*. IEEE Computer Society Press, June 2002. (Cited on pages 63, 65, 79, and 81.)

[MHH$^+$05] Sarah E. Michalak, Kevin W. Harris, Nicolas W. Hengartner, Bruce E. Takala, and Stephen A. Wender. Predicting the number of fatal soft errors in Los Alamos National Laboratory's ASC Q supercomputer. *IEEE Transactions on Device and Materials Reliability*, 5(3):329–335, September 2005. doi: 10.1109/TDMR.2005.855685. (Cited on page 27.)

[MHZA03] Jose Maiz, Scott Hareland, Kevin Zhang, and Patrick Armstrong. Characterization of multi-bit soft error events in advanced SRAMs. In *Proceedings* *of the IEEE International Electron Devices Meeting (IEDM '03)*, pages 21.4.1–21.4.4. IEEE Computer Society Press, December 2003. doi: 10.1109/IEDM.2003.1269335. (Cited on page 28.)

[Mit98] Melanie Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, Cambridge, MA, USA, 1998. (Cited on page 159.)

[Mit14] Sparsh Mittal. A survey of techniques for improving energy efficiency in embedded computing systems. *International Journal of Computer Aided Engineering and Technology*, 6(4):440–459, 2014. doi: 10.1504/IJCAET.2014.065419. (Cited on page 18.)

[MKGT92] Seyed-Ghassem Miremadi, Johan Karlsson, Ulf Gunneflo, and Jan Torin. Two software techniques for on-line error detection. In *Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing (FTCS '92)*, pages 328–335. IEEE Computer Society Press, July 1992. doi: 10.1109/FTCS.1992.243568. (Cited on pages 39, 40, 43, and 62.)

[MKT$^+$09] Michail Maniatakos, Naghmeh Karimi, Chandra Tirumurti, Abhijit Jas, and Yiorgos Makris. Instruction-level impact comparison of RT- vs. gate-level faults in a modern microprocessor controller. In *Proceedings of the 27th IEEE VLSI Test Symposium (VTS '09)*, pages 9–14. IEEE Computer Society Press, May 2009. doi: 10.1109/VTS.2009.32. (Cited on page 58.)

[MKT$^+$11] Michail Maniatakos, Naghmeh Karimi, Chandra Tirumurti, Abhijit Jas, and Yiorgos Makris. Instruction-level impact analysis of low-level faults in a modern microprocessor controller. *IEEE Transactions on Computers*, 60(9):1260–1273, September 2011. doi: 10.1109/TC.2010.60. (Cited on page 58.)

[MRMS94] Henrique S. Madeira, Mário Rela, Francisco Moreira, and João Gabriel Silva. RIFLE: A general purpose pin-level fault injector. In Klaus Echtle, Dieter Hammer, and David Powell, editors, *Proceedings of the 1st European Dependable Computing Conference (EDCC '94)*, pages 197–216. Springer-Verlag, 1994. doi: 10.1007/3-540-58426-9_132. (Cited on page 62.)

[MS08] Darek Mihocka and Stanislav Shwartsman. Virtualization without direct execution or jitting: Designing a portable virtual machine infrastructure. In *1st Workshop on Architectural and Microarchitectural Support for Binary Translation (AMAS-BT '08)*, June 2008. (Cited on pages 87 and 93.)

[MSB11] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing.* John Wiley & Sons, Inc., New York, NY, USA, 3rd edition, 2011. (Cited on page 51.)

[MT90] Colin M. Maunder and Rodham E. Tulloss. *The Test Access Port and Boundary-Scan Architecture.* IEEE Computer Society Press, 1990. (Cited on pages 7, 63, and 169.)

[MT95] Seyed-Ghassem Miremadi and Jan Torin. Evaluating processor-behavior and three error-detection mechanisms using physical fault-injection. *IEEE Transactions on Reliability*, 44(3):441–454, September 1995. doi: 10.1109/24.406580. (Cited on pages 43, 60, and 62.)

[Muk08] Shubu Mukherjee. *Architecture Design for Soft Errors.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008. (Cited on pages 1, 2, 17, 18, 20, 21, 22, 23, 27, 28, 29, 30, 31, 32, 33, 36, 37, 41, 62, and 197.)

[MW79] Timothy C. May and Murray H. Woods. Alpha-particle-induced soft errors in dynamic memories. *IEEE Transactions on Electron Devices*, 26(1):2–9, January 1979. doi: 10.1109/T-ED.1979.19370. (Cited on pages 4, 27, and 57.)

[MWE+03a] Shubendu S. Mukherjee, Christopher T. Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. Measuring architectural vulnerability factors. *IEEE Micro*, 23(6):70–75, November 2003. doi: 10.1109/MM.2003.1261389. (Cited on pages 54, 76, and 203.)

[MWE+03b] Shubhendu S. Mukherjee, Christopher T. Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th Annual IEEE/ACM Symposium on Microarchitecture*, pages 29–40, Los Alamitos, CA, USA, December 2003. IEEE Computer Society Press. doi: 10.1109/MICRO.2003.1253181. (Cited on pages 53, 54, 76, 203, and 205.)

[MWKM15] Justin Meza, Qiang Wu, Sanjeev Kumar, and Onur Mutlu. Revisiting memory errors in large-scale production data centers: Analysis and modeling of new trends from the field. In *Proceedings of the 45th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '15)*, pages 415–426. IEEE Computer Society Press, June 2015. doi: 10.1109/DSN.2015.57. (Cited on page 56.)

[NCDM13] Roberto Natella, Domenico Cotroneo, João A. Durães, and Henrique S. Madeira. On fault representativeness of software fault injection. *IEEE Transactions on Software Engineering*, 39(1):80–96, January 2013. doi: 10.1109/TSE.2011.124. (Cited on pages 4 and 58.)

[NDO11] Edmund B. Nightingale, John R. Douceur, and Vince Orgovan. Cycles, cells and platters: An empirical analysis of hardware failures on a million consumer PCs. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2011 (EuroSys '11)*, pages 343–356, New York, NY, USA, April 2011. ACM Press. doi: 10.1145/1966445.1966477. (Cited on page 27.)

[Neu10] Jens Neuhalfen. Proactive memory error detection for the Linux kernel. Diplomarbeit, TU Dortmund, June 2010. (Cited on page 14.)

[Nic10] Michael Nicolaidis. *Soft Errors in Modern Electronic Systems*, volume 41 of *Frontiers in Electronic Testing*. Springer-Verlag, 1st edition, 2010. doi: 10.1007/978-1-4419-6993-4. (Cited on pages 23 and 31.)

[NIS] *NIST/SEMATECH e-Handbook of Statistical Methods, http://www.itl.nist.gov/div898/handbook/*. Retrieved 2015-05-19. (Cited on pages 62 and 197.)

[Nor96] Eugene Normand. Single event upset at ground level. *IEEE Transactions on Nuclear Science*, 43(6):2742–2750, 1996. (Cited on pages 27 and 57.)

[NSV04] Bogdan Nicolescu, Yvon Savaria, and Raoul Velazco. Software detection mechanisms providing full coverage against single bit-flip faults. *IEEE Transactions on Nuclear Science*, 51(6):3510–3518, December 2004. doi: 10.1109/TNS.2004.839110. (Cited on pages 10, 199, and 200.)

[NTA78] Ravindra Nair, Satish M. Thatte, and Jacob A. Abraham. Efficient algorithms for testing semiconductor random-access memories. *IEEE Transactions on Computers*, C-27(6):572–576, June 1978. doi: 10.1109/TC.1978.1675150. (Cited on page 56.)

[NV03] Bogdan Nicolescu and Raoul Velazco. Detecting soft errors by a purely software approach: method, tools and experimental results. In *Embedded Software for SoC*, pages 39–51. Springer-Verlag, 2003. (Cited on page 3.)

[NVSR01] Bogdan Nicolescu, Raoul Velazco, and Matteo Sonza Reorda. Effectiveness and limitations of various software techniques for "soft error" detection: A comparative study. In *7th International On-Line Testing Workshop*, pages 172–177. IEEE Computer Society Press, July 2001. doi: 10.1109/OLT.2001.937838. (Cited on page 41.)

[NX06] Vijaykrishnan Narayanan and Yuan Xie. Reliability concerns in embedded system designs. *IEEE Computer*, 39(1):118–120, 2006. (Cited on page 1.)

[OR95] Joakim Ohlsson and Marcus Rimén. Implicit signature checking. In *Proceedings of the 25th Annual International Symposium on Fault-Tolerant Computing (FTCS '95)*, pages 218–227. IEEE Computer Society Press, June 1995. doi: 10.1109/FTCS.1995.466976. (Cited on page 44.)

[ORQ$^+$14] Daniel A. G. Oliveira, Paolo Rech, Heather M. Quinn, Thomas D. Fairbanks, Laura Monroe, Sarah E. Michalak, Christine Anderson-Cook, Philippe O. A. Navaux, and Luigi Carro. Modern GPUs radiation sensitivity evaluation and mitigation through duplication with comparison. *IEEE Transactions on Nuclear Science*, 61(6):3115–3122, December 2014. doi: 10.1109/TNS.2014.2362014. (Cited on page 60.)

[ORT$^+$96] Timothy J. O'Gorman, John M. Ross, Allen H. Taber, James F. Ziegler, Hans P. Muhlfeld, Charles J. Montrose, Huntington W. Curtis, and James L. Walsh. Field testing for cosmic ray soft errors in semiconductor memories. *IBM Journal of Research and Development*, 40(1):41–50, January 1996. doi: 10.1147/rd.401.0041. (Cited on page 27.)

[OSM02] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, 51(1):63–75, March 2002. doi: 10.1109/24.994913. (Cited on pages 3, 10, and 43.)

[Pan] Pandaboard homepage. http://pandaboard.org. (Cited on page 86.)

[Pan07] Bernd Panzer-Steindel. Data integrity. Technical report, CERN, Geneve, Switzerland, 2007. (Cited on page 27.)

[PB61] W. Wesley Peterson and David T. Brown. Cyclic codes for error detection. *Proceedings of the IRE*, 49(1):228–235, January 1961. doi: 10.1109/JRPROC.1961.287814. (Cited on page 41.)

[PCZ$^+$08] Andrea Pellegrini, Kypros Constantinides, Dan Zhang, Shobana Sudhakar, Valeria Bertacco, and Todd Austin. CrashTest: A fast high-fidelity FPGA-based resiliency analysis framework. In *Proceedings of the 26th IEEE International Conference on Computer Design (ICCD '08)*, pages 363–370. IEEE Computer Society Press, October 2008. doi: 10.1109/ICCD.2008.4751886. (Cited on pages 66 and 73.)

[Pf036] Georg Pfotzer. Dreifachkoinzidenzen der Ultrastrahlung aus vertikaler Richtung in der Stratosphäre. *Zeitschrift für Physik*, 102(1–2):41–58, 1936. doi: 10.1007/BF01336830. (Cited on page 29.)

[PGZ08] Karthik Pattabiraman, Vinod Grover, and Benjamin G. Zorn. Samurai: Protecting critical data in unsafe languages. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (EuroSys '08)*, pages 219–232, New York, NY, USA, 2008. ACM Press. doi: 10.1145/1352592.1352616. (Cited on pages 3 and 42.)

[Pig05] Michel Pignol. How to cope with SEU/SET at system level? In *Proceedings of the 11th International On-Line Testing Symposium (IOLTS '05)*, pages 315–318. IEEE Computer Society Press, July 2005. doi: 10.1109/IOLTS.2005.34. (Cited on page 43.)

[PKI05] Karthik Pattabiraman, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Application-based metrics for strategic placement of detectors. In *Proceedings of the 11th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC '05)*, pages 75–82. IEEE Computer Society Press, 2005. (Cited on page 11.)

[PKI07] Karthik Pattabiraman, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Automated derivation of application-aware error detectors using static analysis. In *Proceedings of the 13th International On-Line Testing Symposium (IOLTS '07)*, pages 211–216. IEEE Computer Society Press, July 2007. doi: 10.1109/IOLTS.2007.21. (Cited on page 11.)

[PKI11] Karthik Pattabiraman, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Automated derivation of application-aware error detectors using static analysis: The trusted illiac approach. *IEEE Transactions on Dependable and Secure Computing*, 8(1):44–57, January 2011. doi: 10.1109/TDSC.2009.23. (Cited on page 11.)

[PMAC95] David Powell, Elaine Martins, Jean Arlat, and Yves Crouzet. Estimators for fault tolerance coverage evaluation. *IEEE Transactions on Computers*, 44(2):261–274, February 1995. doi: 10.1109/12.364537. (Cited on pages 10, 53, and 188.)

[PMW96] Divya Prasad, John McDermid, and Ian Wand. Dependability terminology: Similarities and differences. *Aerospace and Electronic Systems Magazine*, 11(1):14–21, January 1996. doi: 10.1109/62.484145. (Cited on pages 21 and 24.)

[PNKI08] Karthik Pattabiraman, Nithin M. Nakka, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. SymPLFIED: Symbolic program-level fault injection and error detection framework. In *Proceedings of the 38th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '08)*, pages 472–481, June 2008. doi: 10.1109/DSN.2008.4630118. (Cited on pages 11 and 67.)

[PNKI13] Karthik Pattabiraman, Nithin M. Nakka, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. SymPLFIED: Symbolic program-level fault injection and error detection framework. *IEEE Transactions on Computers*, 62(11):2292–2307, November 2013. doi: 10.1109/TC.2012.219. (Cited on pages 11 and 67.)

[PPS$^+$10] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th International Symposium on Code Generation and Optimization (CGO '10)*, pages 2–11. ACM Press, April 2010. doi: 10.1145/1772954.1772958. (Cited on page 170.)

[PRSR98] Paolo Prinetto, Maurizio Rebaudengo, and Matteo Sonza Reorda. Exploiting the background debugging mode in a fault injection system. In *Proceedings of the IEEE International Computer Performance and Dependability Symposium (IPDS '98)*, page 277. IEEE Computer Society Press, September 1998. doi: 10.1109/IPDS.1998.707736. (Cited on pages 58, 63, 79, and 172.)

[PRSRV00a] B. Parrotta, Maurizio Rebaudengo, Matteo Sonza Reorda, and Massimo Violante. New techniques for accelerating fault injection in VHDL descriptions. In *Proceedings of the 6th IEEE International On-Line Testing Workshop (IOLTW '00)*, pages 61–66. IEEE Computer Society Press, July 2000. doi: 10.1109/OLT.2000.856613. (Cited on pages 10 and 73.)

[PRSRV00b] B. Parrotta, Maurizio Rebaudengo, Matteo Sonza Reorda, and Massimo Violante. Speeding-up fault injection campaigns in VHDL models. In *Proceedings of the 19th International Conference on Computer Safety, Reliability and Security (SAFECOMP '00)*, pages 27–36. Springer-Verlag, October 2000. doi: 10.1007/3-540-40891-6_3. (Cited on pages 10 and 73.)

[PSC$^+$11] Karthik Pattabiraman, Giacinto Paolo Saggese, Daniel Chen, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Automated derivation of application-specific error detectors using dynamic analysis. *IEEE Transactions on Dependable and Secure Computing*, 8(5):640–655, September/October 2011. doi: 10.1109/TDSC.2010.19. (Cited on page 11.)

[PSDC07] Stefan Potyra, Volkmar Sieh, and Mario Dal Cin. Evaluating fault-tolerant system designs using FAUmachine. In *Proceedings of the 2nd International Workshop on Engineering Fault Tolerant Systems (EFTS '07)*, New York, NY, USA, 2007. ACM Press. doi: 10.1145/1316550.1316559. (Cited on pages 6, 58, 66, 78, 81, and 85.)

[PTAB14] Konstantinos Parasyris, Georgios Tziantzoulis, Christos D. Antonopoulos, and Nikolaos Bellas. GemFI: A fault injection tool for studying the behavior of applications on unreliable substrates. In *Proceedings of the 44th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '14)*, pages 622–629, Washington, DC, USA, June 2014. IEEE Computer Society Press. (Cited on pages 6, 10, 51, 67, 73, 78, 81, and 85.)

[PWSF15] Thorsten Piper, Stefan Winter, Neeraj Suri, and Thomas E. Fuhrman. On the effective use of fault injection for the assessment of AUTOSAR safety mechanisms. In *Proceedings of the 11th European Dependable Computing Conference (EDCC '15)*, pages 85–96. IEEE Computer Society Press, September 2015. doi: 10.1109/EDCC.2015.14. (Cited on page 58.)

[QBRB13] Heather M. Quinn, Dolores A. Black, William H. Robinson, and Stephen P. Buchner. Fault simulation and emulation tools to augment radiation-hardness assurance testing. *IEEE Transactions on Nuclear Science*, 60(3):2119–2142, June 2013. doi: 10.1109/TNS.2013.2259503. (Cited on pages 61, 197, and 213.)

[RAA02] Manuel Rodríguez, Arnaud Albinet, and Jean Arlat. MAFALDA-RT: A tool for dependability assessment of real-time systems. In *Proceedings of the 32nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '02)*, pages 267–272. IEEE Computer Society Press, June 2002. doi: 10.1109/DSN.2002.1028909. (Cited on pages 59 and 65.)

[Rad13] Lars Rademacher. FailPanda: Fehlerinjektionsexperimente auf einer eingebetteten ARM-Plattform. Master thesis, Technische Universität Dortmund, December 2013. (Cited on pages 14 and 208.)

[Rat04] David Ratter. FPGAs on Mars. *Xilinx Xcell Journal*, pages 8–11, 2004. (Cited on pages 37 and 38.)

[Rat05] Dominic Rath. *OpenOCD: Open On-Chip Debugging*. Diploma thesis, FH Augsburg, July 2005. (Cited on pages 88, 96, and 97.)

[RBQ96] Christophe Rabéjac, Jean-Paul Blanquart, and Jean-Pierre Queille. Executable assertions and timed traces for on-line software error detection. In *Proceedings of the 26th Annual International Symposium on Fault-Tolerant Computing (FTCS '96)*, pages 138–147, June 1996. doi: 10.1109/FTCS.1996.534602. (Cited on pages 40 and 43.)

[RC13] Paolo Rech and Luigi Carro. Experimental evaluation of gpus radiation sensitivity and algorithm-based fault tolerance efficiency. In *Proceedings of the 19th International On-Line Testing Symposium (IOLTS '13)*, pages 244–247. IEEE Computer Society Press, July 2013. doi: 10.1109/IOLTS.2013.6604091. (Cited on page 60.)

[RCA+06] George A. Reis, Jonathan Chang, David I. August, Robin Cohn, and Shubhendu S. Mukherjee. Configurable transient fault detection via dynamic binary translation. In *Proceedings of the 2nd Workshop on Architectural Reliability (WAR '06)*, 2006. (Cited on page 203.)

[RCA07] George A. Reis, Jonathan Chang, and David I. August. Automatic instruction-level software-only recovery. *IEEE Micro*, 27(1):36–47, January 2007. doi: 10.1109/MM.2007.4. (Cited on page 203.)

[RCE02] Michael Redeker, Bruce F. Cockburn, and Duncan G. Elliott. An investigation into crosstalk noise in DRAM structures. In *Proceedings of the 2002 IEEE International Workshop on Memory Technology, Design and Testing*, MTDT '02, pages 123–129. IEEE Computer Society Press, 2002. doi: 10.1109/MTDT.2002.1029773. (Cited on page 31.)

[RCV+05a] George A. Reis, Jonathan Chang, Neil Vachharajani, Shubhendu S. Mukherjee, Ram Rangan, and David I. August. Design and evaluation of hybrid fault-detection systems. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA '05)*, pages 148–159. IEEE Computer Society Press, June 2005. doi: 10.1109/ISCA.2005.21. (Cited on pages 37 and 203.)

[RCV+05b] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. SWIFT: Software implemented fault tolerance. In *Proceedings of the 3rd International Symposium on Code Generation and Optimization (CGO '05)*, pages 243–254, Los Alamitos, CA, USA, 2005. IEEE Computer Society Press. doi: 10.1109/CGO.2005.34. (Cited on pages 3, 10, and 43.)

[RCV+05c] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, David I. August, and Shubhendu S. Mukherjee. Software-controlled fault tolerance. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2(4):366–396, December 2005. doi: 10.1145/1113841.1113843. (Cited on pages 38 and 39.)

[Rec15] Paolo Rech. Personal correspondence, 2014–2015. (Cited on page 61.)

[RKK+08] Pradeep Ramachandran, Prabhakar Kudva, Jeffrey Kellington, John Schumann, and Pia Sanda. Statistical fault injection. In *Proceedings of the 38th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '08)*, pages 122–127, Washington, DC, USA, June 2008. IEEE Computer Society Press. (Cited on pages 53, 68, and 150.)

[RLX+98] Gregory L. Ries, Myeong S. Lee, Yuxiao Xiao, Zbigniew Kalbarczyk, Jaqdish U. Patel, and Ravishankar K. Iyer. Hierarchical approach to accurate fault modeling for system evaluation. In *Proceedings of the IEEE International Computer Performance and Dependability Symposium (IPDS '98)*, pages 249–258. IEEE Computer Society Press, September 1998. doi: 10.1109/IPDS.1998.707727. (Cited on page 58.)

[RM00] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA '00)*, pages 25–36, New York, NY, USA, 2000. ACM Press. doi: 10.1145/339647.339652. (Cited on page 38.)

[RMCJ13] Felipe Restrepo-Calle, Antonio Martínez-Álvarez, Sergio Cuenca-Asensi, and Antonio Jimeno-Morenilla. Selective SWIFT-R. *Journal of Electronic Testing*, 29(6):825–838, 2013. doi: 10.1007/s10836-013-5416-6. (Cited on page 3.)

[RMM04] Amir Rajabzadeh, Seyed-Ghassem Miremadi, and Mirzad Mohandespour. Experimental evaluation of master/checker architecture using power supply- and software-based fault injection. In *Proceedings of the 10th International On-Line Testing Symposium (IOLTS '04)*, Washington, DC, USA, 2004. IEEE Computer Society Press. doi: 10.1109/IOLTS.2004.21. (Cited on page 62.)

[Rot99] Eric Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing (FTCS '99)*, pages 84–91, Washington, DC, USA, June 1999. IEEE Computer Society Press. doi: 10.1109/FTCS.1999.781037. (Cited on page 38.)

[RPWD05] Daniele Radaelli, Helmut Puchner, Skip Wong, and Sabbas Daniel. Investigation of multi-bit upsets in a 150 nm technology SRAM device. *IEEE Transactions on Nuclear Science*, 52(6):2433–2437, December 2005. doi: 10.1109/TNS.2005.860675. (Cited on page 28.)

[RSFA99] Manuel Rodríguez, Frédéric Salles, Jean-Charles Fabre, and Jean Arlat. MAFALDA: Microkernel assessment by fault injection and design aid. In *Proceedings of the 3rd European Dependable Computing Conference (EDCC '99)*, pages 143–160. Springer-Verlag, September 1999. doi: 10.1007/3-540-48254-7_11. (Cited on page 65.)

[RSKH11] Semeen Rehman, Muhammad Shafique, Florian Kriebel, and Jörg Henkel. Reliable software for unreliable hardware: Embedded code generation aiming at reliability. In *Proceedings of the 9th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '11)*, pages 237–246, Taipei, Taiwan, October 2011. (Cited on pages 76 and 203.)

[RSR99] Maurizio Rebaudengo and Matteo Sonza Reorda. Evaluating the fault tolerance capabilities of embedded systems via BDM. In *Proceedings of the 17th IEEE VLSI Test Symposium (VTS '99)*, pages 452–457. IEEE Computer Society Press, April 1999. doi: 10.1109/VTEST.1999.766703. (Cited on pages 7, 63, 172, and 173.)

[RSRVT01] Maurizio Rebaudengo, Matteo Sonza Reorda, Massimo Violante, and Marco Torchiano. A source-to-source compiler for generating dependable software. In *Proceedings of the 1st IEEE International Workshop on Source Code Analysis and Manipulation*, pages 33–42. IEEE Computer Society Press, 2001. doi: 10.1109/SCAM.2001.972664. (Cited on pages 10 and 199.)

[RTSM11] Daniele Rossi, Nicola Timoncini, Michael Spica, and Cecilia Metra. Error correcting code analysis for cache memory high reliability and performance. In *Proceedings of the 2011 Conference on Design, Automation & Test in Europe (DATE '11)*, March 2011. doi: 10.1109/DATE.2011.5763257. (Cited on pages 2 and 37.)

[Sai77] Sabina H. Saib. Executable assertions – an aid to reliable software. In *Proceedings of the 11th Asilomar Conference on Circuits, Systems and Computers*, pages 277–281. IEEE Computer Society Press, November 1977. doi: 10.1109/ACSSC.1977.748932. (Cited on page 40.)

[Sai05] Yasushi Saito. Jockey: A user-space library for record-replay debugging. In *Proceedings of the 6th International Symposium on Automated and Analysis-Driven Debugging (AADEBUG '05)*, pages 69–76. ACM Press, September 2005. doi: 10.1145/1085130.1085139. (Cited on page 170.)

[SAIC+99] Timothy J. Slegel, Robert M. Averill III, Mark A. Check, Bruce C. Giamei, Barry W. Krumm, Christopher A. Krygowski, Wen H. Li, John S. Liptay, John D. MacDougall, Thomas J. McPherson, Jennifer A. Navarro, Eric M. Schwarz, Kevin Shum, and Charles F. Webb. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19(2):12–23, March 1999. doi: 10.1109/40.755464. (Cited on pages 37 and 38.)

[SAJK15] Behrooz Sangchoolie, Fatemeh Ayatolahi, Roger Johansson, and Johan Karlsson. A comparison of inject-on-read and inject-on-write in ISA-level fault injection. In *Proceedings of the 11th European Dependable Computing Conference (EDCC '15)*, pages 178–189. IEEE Computer Society Press, September 2015. doi: 10.1109/EDCC.2015.24. (Cited on page 58.)

[SBD+16] Thiago Santini, Christoph Borchert, Christian Dietrich, **Horst Schirmeier**, Martin Hoffmann, Olaf Spinczyk, Daniel Lohmann, Flávio Rech Wagner, and Paolo Rech. Evaluating the radiation reliability of dependability-oriented real-time operating systems. In *Proceedings of the 12th Workshop on Silicon Errors in Logic – System Effects (SELSE '16)*, March 2016. (Cited on pages vii and 213.)

[SBK10] Daniel Skarin, Raul Barbosa, and Johan Karlsson. GOOFI-2: A tool for experimental dependability assessment. In *Proceedings of the 40th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '10)*, pages 557–562, Washington, DC, USA, June/July 2010. IEEE Computer Society Press. doi: 10.1109/DSN.2010.5544265. (Cited on pages 58, 59, 63, 64, 65, 69, 78, 79, and 172.)

[SBM+09] Alex Shye, Joseph Blomstedt, Tipp Moseley, Vijay Janapa Reddi, and Daniel A. Connors. PLR: A software approach to transient fault tolerance for multicore architectures. *IEEE Transactions on Dependable and Secure Computing*, 6(2):135–148, April 2009. doi: 10.1109/TDSC.2008.62. (Cited on page 44.)

[SBS14] **Horst Schirmeier**, Christoph Borchert, and Olaf Spinczyk. Rapid fault-space exploration by evolutionary pruning. In *Proceedings of the 33rd International Conference on Computer Safety, Reliability and Security (SAFECOMP '14)*, Lecture Notes in Computer Science, pages 17–32. Springer-Verlag, September 2014. doi: 10.1007/978-3-319-10506-2_2. (Cited on pages vi, 8, 9, 14, 48, 73, 145, 150, 154, 164, and 209.)

[SBS15] **Horst Schirmeier**, Christoph Borchert, and Olaf Spinczyk. Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors. In *Proceedings of the 45th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '15)*, pages 319–330, Washington, DC, USA, June 2015. IEEE Computer Society Press. doi: 10.1109/DSN.2015.44. (Cited on pages vi, 3, 8, 14, 15, 48, 145, 186, 194, and 209.)

[Sch97] Robert R. Schaller. Moore's law: past, present and future. *IEEE Spectrum*, 34(6):52–59, June 1997. doi: 10.1109/6.591665. (Cited on pages 1, 18, and 30.)

[Sch11] Ute Schiffel. *Hardware Error Detection Using AN-Codes.* Dissertation, Technische Universität Dresden, June 2011. (Cited on pages 42 and 64.)

[Sch15a] Gerhard Schönfelder. FIES: A fault injection framework for the evaluation of self-tests. Master thesis, TU Graz, January 2015. (Cited on page 67.)

[Sch15b] Simon Schröder. Entwicklung und Bewertung von Advice für vordefinierte Operatoren in AspectC++. Bachelor thesis, Technische Universität Dortmund, February 2015. (Cited on page 208.)

[Sch15c] Simon Schuster. Ein Kontrollflussüberwachungsdienst für KESO Anwendungen. Bachelor thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, June 2015. (Cited on page 208.)

[Sch16] **Horst Schirmeier**. *Efficient Fault-Injection-based Assessment of Software-Implemented Hardware Fault Tolerance.* Dissertation, Technische Universität Dortmund, July 2016. (Cited on page 208.)

[SDB+15] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. Memory errors in modern systems: The good, the bad, and the ugly. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*, pages 297–310, New York, NY, USA, 2015. ACM Press. doi: 10.1145/2694344.2694348. (Cited on page 28.)

[Sel15] Marcel Sellung. Vergleich von Hardwarefehlermodellen bei Fehlerinjektions-Experimenten. Bachelor thesis, Technische Universität Dortmund, March 2015. (Cited on page 208.)

[SFKI00] David T. Stott, Benjamin Floering, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. NFTAPE: A framework for assessing dependability in distributed systems with lightweight fault injectors. In *Proceedings of the IEEE International Computer Performance and Dependability Symposium (IPDS '00)*, pages 91–100. IEEE Computer Society Press, March 2000. doi: 10.1109/IPDS.2000.839467. (Cited on page 65.)

[SG06] Bianca Schroeder and Garth A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proceedings of the 36th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '06)*, pages 249–258, Washington, DC, USA, June 2006. IEEE Computer Society Press. doi: 10.1109/DSN.2006.5. (Cited on pages 27 and 28.)

[SG10] Bianca Schroeder and Garth A. Gibson. A large-scale study of failures in high-performance computing systems. *IEEE Transactions on Dependable and Secure Computing*, 7(4):337–350, October 2010. doi: 10.1109/TDSC.2009.4. (Cited on pages 2, 27, and 28.)

[SHD+15] **Horst Schirmeier**, Martin Hoffmann, Christian Dietrich, Michael Lenz, Daniel Lohmann, and Olaf Spinczyk. FAIL*: An open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance. In *Proceedings of the 11th European Dependable Computing Conference (EDCC '15)*, pages 245–255. IEEE Computer Society Press, September 2015. doi: 10.1109/EDCC.2015.28. (Cited on pages vii, 14, 48, 84, 88, and 108.)

[SHK+11] **Horst Schirmeier**, Martin Hoffmann, Rüdiger Kapitza, Daniel Lohmann, and Olaf Spinczyk. Revisiting fault-injection experiment-platform architectures. In *Proceedings of the 17th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC '11)*, pages 284–285, Pasadena, CA, USA, December 2011. IEEE Computer Society Press. Fast abstract. doi: 10.1109/PRDC.2011.46. (Cited on pages v, 7, 14, and 48.)

[SHK+12] **Horst Schirmeier**, Martin Hoffmann, Rüdiger Kapitza, Daniel Lohmann, and Olaf Spinczyk. FAIL*: Towards a versatile fault-injection experiment framework. In Gero Mühl, Jan Richling, and Andreas Herkersdorf, editors, *25th International Conference on Architecture of Computing Systems (ARCS '12), Workshop Proceedings*, volume 200 of *Lecture Notes in Informatics*, pages 201–210. German Society of Informatics, March 2012. (Cited on pages v, 14, 48, 84, and 208.)

[SHLR+09] Siva Kumar Sastry Hari, Man-Lap Li, Pradeep Ramachandran, Byn Choi, and Sarita V. Adve. mSWAT: Low-cost hardware fault detection and diagnosis for multicore systems. In *Proceedings of the 42nd IEEE/ACM International*

Symposium on Microarchitecture (MICRO '09), pages 122–132, New York, NY, USA, 2009. ACM Press. doi: 10.1145/1669112.1669129. (Cited on page 3.)

[SJAP97] D. Todd Smith, Barry W. Johnson, Nikos Andrianos, and Joseph A. Profeta, III. A variance-reduction technique via fault-expansion for fault-coverage estimation. *IEEE Transactions on Reliability*, 46(3):366–374, September 1997. doi: 10.1109/24.664008. (Cited on page 165.)

[SJPB95] D. Todd Smith, Barry W. Johnson, Joseph A. Profeta, III, and Daniele G. Bozzolo. A method to determine equivalent fault classes for permanent and transient faults. In *Proceedings of the Annual Reliability and Maintainability Symposium*, pages 418–424. IEEE Computer Society Press, January 1995. doi: 10.1109/RAMS.1995.513278. (Cited on pages 8 and 70.)

[SK08a] Daniel Skarin and Johan Karlsson. Software implemented detection and recovery of soft errors in a brake-by-wire system. In *Proceedings of the 7th European Dependable Computing Conference (EDCC '08)*, pages 145–154, Washington, DC, USA, 2008. IEEE Computer Society Press. doi: 10.1109/EDCC-7.2008.24. (Cited on pages 40 and 193.)

[SK08b] Vilas Sridharan and David R. Kaeli. Quantifying software vulnerability. In *Proceedings of the Workshop on Radiation Effects and Fault Tolerance in Nanometer Technologies (WREFT '08)*, pages 323–328, New York, NY, USA, 2008. ACM Press. doi: 10.1145/1366224.1366225. (Cited on pages 11, 54, and 203.)

[SK09a] Vilas Sridharan and David R. Kaeli. The effect of input data on program vulnerability. In *Proceedings of the 5th Workshop on Silicon Errors in Logic – System Effects (SELSE '09)*, March 2009. (Cited on page 52.)

[SK09b] Vilas Sridharan and David R. Kaeli. Eliminating microarchitectural dependency from architectural vulnerability. In *Proceedings of the 15th IEEE International Symposium on High Performance Computer Architecture (HPCA '09)*, pages 117–128. IEEE Computer Society Press, February 2009. doi: 10.1109/HPCA.2009.4798243. (Cited on pages 11, 54, 76, and 203.)

[Ska10] Daniel Skarin. *On Fault Injection-Based Assessment of Safety-Critical Systems*. PhD thesis, Chalmers University of Technology, 2010. (Cited on pages 63, 64, 65, and 172.)

[SKK$^+$02] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings of the 32nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '02)*, pages 389–398, June 2002. doi: 10.1109/DSN.2002.1028924. (Cited on page 1.)

[SKLS11] **Horst Schirmeier**, Rüdiger Kapitza, Daniel Lohmann, and Olaf Spinczyk. DanceOS: Towards dependability aspects in configurable embedded operating systems. In Alex Orailoglu, editor, *Proceedings of the 3rd HiPEAC Workshop on Design for Reliability (DFR '11)*, pages 21–26, Heraklion, Greece, January 2011. (Cited on pages v and 118.)

[SKSE13] **Horst Schirmeier**, Ingo Korb, Olaf Spinczyk, and Michael Engel. Efficient online memory error assessment and circumvention for Linux with RAMpage. *International Journal of Critical Computer-Based Systems*, 4(3):227–247, 2013. Special Issue on PRDC 2011 Dependable Architecture and Analysis. doi: 10.1504/IJCCBS.2013.058397. (Cited on pages v, 14, 84, 87, 99, 101, 145, and 209.)

[SL07] Olaf Spinczyk and Daniel Lohmann. The design and implementation of AspectC++. *Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software*, 20(7):636–651, 2007. doi: 10.1016/j.knosys.2007.05.004. (Cited on pages 45, 93, and 94.)

[SL12] Vilas Sridharan and Dean Liberty. A study of DRAM failures in the field. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '12)*, pages 76:1–76:11, Los Alamitos, CA, USA, November 2012. IEEE Computer Society Press. doi: 10.1109/SC.2012.13. (Cited on pages 28, 37, and 57.)

[Sla11] Charles Slayman. Soft error trends and mitigation techniques in memory devices. In *Proceedings of the Annual Reliability and Maintainability Symposium (RAMS)*. IEEE Computer Society Press, January 2011. doi: 10.1109/RAMS.2011.5754515. (Cited on pages 27, 28, 30, and 48.)

[SM98] Philip P. Shirvani and Edward J. McCluskey. Fault-tolerant systems in a space environment: The CRC ARGOS project. Technical Report 98-2, Center for Reliable Computing, Stanford University, Stanford, CA, USA, December 1998. (Cited on page 41.)

[SMF97] John R. Samson, Jr., Wilfrido A. Moreno, and Fernando J. Falquez. Validating fault tolerant designs using laser fault injection (LFI). In *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT '97)*, pages 175–183. IEEE Computer Society Press, October 1997. doi: 10.1109/DFTVS.1997.628323. (Cited on page 61.)

[SMF98] John R. Samson, Jr., Wilfrido A. Moreno, and Fernando J. Falquez. A technique for automated validation of fault tolerant designs using laser fault injection (LFI). In *Proceedings of the 28th Annual International Symposium on Fault-Tolerant Computing (FTCS '98)*, pages 162–167. IEEE Computer Society Press, June 1998. doi: 10.1109/FTCS.1998.689466. (Cited on page 61.)

[SMHW02] Daniel J. Sorin, Milo M. K. Martin, Mark D. Hill, and David A. Wood. SafetyNet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proceedings of the 29th International Symposium on Computer Architecture (ISCA '02)*, pages 123–134. IEEE Computer Society Press, 2002. doi: 10.1109/ISCA.2002.1003568. (Cited on page 37.)

[Smi82] Brian Cantwell Smith. *Procedural Reflection in Programming Languages*. Dissertation, Massachusetts Institute of Technology, February 1982. (Cited on page 45.)

[SMR+07] Alex Shye, Tipp Moseley, Vijay Janapa Reddi, Joseph Blomstedt, and Daniel A. Connors. Using process-level redundancy to exploit multiple cores for transient fault tolerance. In *Proceedings of the 37th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '07)*, pages 297–306. IEEE Computer Society Press, June 2007. doi: 10.1109/DSN.2007.98. (Cited on page 44.)

[SNK+11] **Horst Schirmeier**, Jens Neuhalfen, Ingo Korb, Olaf Spinczyk, and Michael Engel. RAMpage: Graceful degradation management for memory errors in commodity Linux servers. In *Proceedings of the 17th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC '11)*, pages 89–98, Pasadena, CA, USA, December 2011. IEEE Computer Society Press. doi: 10.1109/PRDC.2011.20. (Cited on pages v, 14, 84, 87, 99, and 101.)

[SOM+00] Philip P. Shirvani, Namsuk Oh, Edward J. Mccluskey, D. L. Wood, Michael N. Lovellette, and K. S. Wood. Software-implemented hardware fault tolerance experiments: COTS in space. In *Proceedings of the 30th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '00)*, pages

B56–B57. IEEE Computer Society Press, June 2000. Fast abstract. (Cited on pages 41 and 48.)

[Sor09] Daniel J. Sorin. *Fault Tolerant Computer Architecture.* Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, 1st edition, 2009. doi: 10.2200/S00192ED1V01Y200904CAC005. (Cited on page 23.)

[SPS09] Matthias Sand, Stefan Potyra, and Volkmar Sieh. Deterministic high-speed simulation of complex systems including fault-injection. In *Proceedings of the 39th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '09)*, pages 211–216, Washington, DC, USA, June/July 2009. IEEE Computer Society Press. doi: 10.1109/DSN.2009.5270335. (Cited on pages 6, 66, 78, 81, and 85.)

[SPW09] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: A large-scale field study. In *Proceedings of the 11th International Joint Conference on Measurement and Modeling of Computer Systems,* SIGMETRICS '09, pages 193–204, New York, NY, USA, 2009. ACM. doi: 10.1145/1555349.1555372. (Cited on pages 27 and 28.)

[SPW11] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: A large-scale field study. *Communications of the ACM,* 54(2):100–107, February 2011. doi: 10.1145/1897816.1897844. (Cited on pages 27 and 28.)

[SRCRW15] Thiago Santini, Paolo Rech, Luigi Carro, and Flávio Rech Wagner. Exploiting cache conflicts to reduce radiation sensitivity of operating systems on embedded systems. In *Proceedings of the 2015 International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES '15)*, pages 49–58. IEEE Computer Society Press, October 2015. doi: 10.1109/CASES.2015.7324545. (Cited on pages 1 and 60.)

[SRFA99] Frédéric Salles, Manuel Rodríguez, Jean-Charles Fabre, and Jean Arlat. MetaKernels and fault containment wrappers. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing (FTCS '99)*, pages 22–29. IEEE Computer Society Press, June 1999. doi: 10.1109/FTCS.1999.781030. (Cited on page 65.)

[SRJW14] Aviral Shrivastava, Abhishek Rhisheekesan, Reiley Jeyapaul, and Carole-Jean Wu. Quantitative analysis of control flow checking mechanisms for soft errors. In *Proceedings of the 51st Design Automation Conference (DAC '14)*, pages 13:1–13:6, New York, NY, USA, 2014. ACM Press. doi: 10.1145/2593069.2593195. (Cited on pages 3 and 205.)

[SRN$^+$14] Thiago Santini, Paolo Rech, Gabriel Nazar, Luigi Carro, and Flávio Rech Wagner. Reducing embedded software radiation-induced failures through cache memories. In *Proceedings of the 19th IEEE European Test Symposium (ETS '14)*, May 2014. doi: 10.1109/ETS.2014.6847793. (Cited on pages 60 and 204.)

[SRS14] **Horst Schirmeier**, Lars Rademacher, and Olaf Spinczyk. Smart-hopping: Highly efficient ISA-level fault injection on real hardware. In *Proceedings of the 19th IEEE European Test Symposium (ETS '14)*, pages 69–74. IEEE Computer Society Press, May 2014. doi: 10.1109/ETS.2014.6847803. (Cited on pages vi, 14, 48, 170, 171, 172, 174, 178, and 209.)

[SS94] Michael A. Schuette and John P. Shen. Exploiting instruction-level parallelism for integrated control-flow monitoring. *IEEE Transactions on Computers,* 43(2):129–140, February 1994. doi: 10.1109/12.262118. (Cited on page 43.)

[SSD+13] Vilas Sridharan, Jon Stearley, Nathan DeBardeleben, Sean Blanchard, and Sudhanva Gurumurthi. Feng shui of supercomputer memory: Positional effects in DRAM and SRAM faults. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '13)*, pages 22:1–22:11, New York, NY, USA, November 2013. ACM Press. doi: 10.1145/2503210.2503257. (Cited on pages 28, 37, 48, 56, and 57.)

[SSE+13] Isabella Stilkerich, Michael Strotz, Christoph Erhardt, Martin Hoffmann, Daniel Lohmann, Fabian Scheler, and Wolfgang Schröder-Preikschat. A JVM for soft-error-prone embedded systems. In *Proceedings of the 2013 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '13)*, pages 21–32, June 2013. doi: 10.1145/2465554.2465571. (Cited on page 209.)

[SSM00] Philip P. Shirvani, Nirmal R. Saxena, and Edward J. McCluskey. Software-implemented EDAC protection against SEUs. *IEEE Transactions on Reliability*, 49(3):273–284, September 2000. doi: 10.1109/24.914544. (Cited on pages 41 and 118.)

[SSSF10a] Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. ANB- and ANBDmem-encoding: Detecting hardware errors in software. In *Proceedings of the 29th International Conference on Computer Safety, Reliability and Security (SAFECOMP '10)*, pages 169–182. Springer-Verlag, September 2010. (Cited on pages 41 and 42.)

[SSSF10b] Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. Slice your bug: Debugging error detection mechanisms using error injection slicing. In *Proceedings of the 8th European Dependable Computing Conference (EDCC '10)*, pages 13–22. IEEE Computer Society Press, 2010. (Cited on pages 11, 58, 64, and 81.)

[STB97] Volkmar Sieh, Oliver Tschäche, and Frank Balbach. VERIFY: Evaluation of reliability using VHDL-models with embedded fault descriptions. In *Proceedings of the 27th Annual International Symposium on Fault-Tolerant Computing (FTCS '97)*, pages 32–36, June 1997. doi: 10.1109/FTCS.1997.614074. (Cited on page 66.)

[STE+14] Isabella Stilkerich, Philip Taffner, Christoph Erhardt, Christian Dietrich, Christian Wawersich, and Michael Stilkerich. Team up: Cooperative memory management in embedded systems. In *Proceedings of the 2014 International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES '14)*, pages 10:1–10:10. ACM Press, October 2014. doi: 10.1145/2656106.2656129. (Cited on page 209.)

[Sti16] Isabella Stilkerich. *Cooperative Memory Management in Safety-Critical Embedded Systems.* Dissertation, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2016. (Cited on page 208.)

[Stu15] Tobias Stumpf. How to protect the protector? In *Proceedings of the 45th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '15), Student Forum.* IEEE Computer Society Press, June 2015. (Cited on page 209.)

[SVS+95] Zary Z. Segall, D. Vrsalovic, Daniel P. Siewiorek, D. Ysskin, J. Kownacki, James H. Barton, R. Dancey, A. Robinson, and T. Lin. FIAT – fault injection based automated testing environment. In *Proceedings of the 25th Annual International Symposium on Fault-Tolerant Computing (FTCS '95)*, June 1995. doi: 10.1109/FTCSH.1995.532663. (Cited on page 64.)

[SZR03] Luís Santos and Mário Zenha Rela. Constraints on the use of boundary-scan for fault injection. In *Proceedings of the 1st Latin-American Symposium on Dependable Computing (LADC '03)*, pages 39–55. Springer-Verlag, 2003. doi: 10.1007/978-3-540-45214-0_6. (Cited on pages 7 and 62.)

[Taf14] Philip Taffner. Design und Implementierung einer fehlertoleranten Speicherbereinigung für die KESO-JVM. Diploma thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, February 2014. (Cited on page 208.)

[TEL95] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA '95)*, pages 392–403, New York, NY, USA, 1995. ACM Press. doi: 10.1145/223982.224449. (Cited on page 38.)

[TI95a] Timothy K. Tsai and Ravishankar K. Iyer. FTAPE: A fault injection tool to measure fault tolerance. In *Proceedings of the 10th Computing in Aerospace Conference*. AIAA, March 1995. (Cited on page 65.)

[TI95b] Timothy K. Tsai and Ravishankar K. Iyer. Measuring fault tolerance with the FTAPE fault injection tool. In *Proceedings of the 8th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation: Quantitative Evaluation of Computing and Communication Systems*, pages 26–40, London, UK, 1995. Springer-Verlag. (Cited on pages 58 and 65.)

[TMB80] David J. Taylor, David E. Morgan, and James P. Black. Redundancy in data structures: Improving software fault tolerance. *IEEE Transactions on Software Engineering*, SE-6(6):585–594, November 1980. doi: 10.1109/TSE.1980.234507. (Cited on page 46.)

[TP13] Anna Thomas and Karthik Pattabiraman. LLFI: An intermediate code level fault injector for soft computing applications. In *Proceedings of the 9th Workshop on Silicon Errors in Logic – System Effects (SELSE '13)*, March 2013. (Cited on pages 32, 58, 64, 71, and 81.)

[Tri02] Kishor S. Trivedi. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. John Wiley & Sons, Inc., New York, NY, USA, second edition, 2002. (Cited on pages 53 and 57.)

[TS09] Peter Tummeltshammer and Andreas Steininger. Power supply induced common cause faults – experimental assessment of potential countermeasures. In *Proceedings of the 39th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '09)*, pages 449–457. IEEE Computer Society Press, June/July 2009. doi: 10.1109/DSN.2009.5270308. (Cited on page 62.)

[TS13] Clemens Terasa and Sibylle Schupp. Annotation-guided soft-error injection. In *Proceedings of the 2nd GI Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES '13)*, Lecture Notes in Informatics. German Society of Informatics, September 2013. (Cited on page 67.)

[Tyr96] Andrew M. Tyrrell. Recovery blocks and algorithm-based fault tolerance. In *Proceedings of the 22nd EUROMICRO Conference*, pages 292–299. IEEE Computer Society Press, September 1996. doi: 10.1109/EURMIC.1996.546394. (Cited on page 39.)

[UHK$^+$12] Peter Ulbrich, Martin Hoffmann, Rüdiger Kapitza, Daniel Lohmann, Wolfgang Schröder-Preikschat, and Reiner Schmid. Eliminating single points of failure in software-based redundancy. In *Proceedings of the 9th European Dependable Computing Conference (EDCC '12)*. IEEE Computer Society Press, May 2012. doi: 10.1109/EDCC.2012.21. (Cited on page 44.)

[Ulb14] Peter Ulbrich. *Ganzheitliche Fehlertoleranz in eingebetteten Softwaresystemen.* Dissertation, Friedrich-Alexander-Universität Erlangen-Nürnberg, August 2014. (Cited on pages 20, 21, 41, and 208.)

[Unz13] Martin Unzner. Implementation of a fault injection framework for Fiasco.OC. Belegarbeit, Technische Universität Dresden, January 2013. (Cited on page 208.)

[VA06] Ramtilak Vemu and Jacob A. Abraham. CEDA: Control-flow error detection through assertions. In *Proceedings of the 12th International On-Line Testing Symposium (IOLTS '06)*, pages 151–158. IEEE Computer Society Press, 2006. doi: 10.1109/IOLTS.2006.14. (Cited on page 204.)

[VAS$^+$05] Jonny Vinter, Joakim Aidemark, Daniel Skarin, Raul Barbosa, Peter Folkesson, and Johan Karlsson. An overview of GOOFI – a generic object-oriented fault injection framework. Technical Report 05-07, Chalmers University of Technology, Gothenburg, Sweden, June 2005. (Cited on pages 63, 64, 78, and 79.)

[VCT$^+$99] Raoul Velazco, Phillipe Cheynet, André Tissot, Jacques Haussy, Jean Lambert, and Robert Ecoffet. Evidences of SEU tolerance for digital implementations of artificial neural networks: one year MPTB flight results. In *Proceedings of the 5th European Conference on Radiation and Its Effects on Components and Systems (RADECS '99)*, pages 565–568. IEEE Computer Society Press, September 1999. doi: 10.1109/RADECS.1999.858648. (Cited on page 39.)

[VG00] Jeffrey M. Voas and Anup K. Ghosh. Software fault injection for survivability. In *Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX '00)*, volume 2, pages 338–346, 2000. doi: 10.1109/DISCEX.2000.821531. (Cited on page 4.)

[Voa97] Jeffrey M. Voas. Fault injection for the masses. *IEEE Transactions on Computers*, 30(12):129–130, December 1997. doi: 10.1109/2.642820. (Cited on page 4.)

[VPC02] T. N. Vijaykumar, Irith Pomeranz, and Karl Cheng. Transient-fault recovery using simultaneous multithreading. In *Proceedings of the 29th International Symposium on Computer Architecture (ISCA '02)*, pages 87–98. IEEE Computer Society Press, 2002. doi: 10.1109/ISCA.2002.1003565. (Cited on page 38.)

[WEL$^+$13] Lucas Wanner, Salma Elmalaki, Liangzhen Lai, Puneet Gupta, and Mani Srivastava. VarEMU: An emulation testbed for variability-aware software. In *Proceedings of the 11th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '13)*, pages 27:1–27:10, Piscataway, NJ, USA, 2013. IEEE Computer Society Press. (Cited on page 67.)

[WEMR04] Christopher Weaver, Joel Emer, Shubhendu S. Mukherjee, and Steven K. Reinhardt. Techniques to reduce the soft error rate of a high-performance microprocessor. In *Proceedings of the 31st International Symposium on Computer Architecture (ISCA '04)*, pages 264–275. IEEE Computer Society Press, June 2004. doi: 10.1109/ISCA.2004.1310780. (Cited on pages 31, 32, and 203.)

[WF07] Ute Wappler and Christof Fetzer. Software encoded processing: Building dependable systems with commodity hardware. In *Proceedings of the 26th International Conference on Computer Safety, Reliability and Security (SAFECOMP '07)*, pages 356–369. Springer-Verlag, 2007. doi: 10.1007/978-3-540-75101-4_34. (Cited on page 41.)

[WH11] Neil H. E. Weste and David Money Harris. *CMOS VLSI Design: A Circuits and Systems Perspective.* Addison-Wesley, Boston, MA, USA, 4th edition, 2011. (Cited on pages 25 and 26.)

[Wik13] Wikipedia. 2009–11 Toyota vehicle recalls. https://en.wikipedia.org/wiki/2009%E2%80%9311_Toyota_vehicle_recalls, 2013. Retrieved 2015-12-06. (Cited on page 2.)

[Wil06] Jeff Wilkinson. Neutron flux calculator. http://seutest.com/cgi-bin/FluxCalculator.cgi, 2006. (Cited on page 29.)

[Win15] Stefan Winter. *On the Utility of Higher Order Fault Models for Fault Injections.* Dissertation, TU Darmstadt, May 2015. (Cited on pages 65 and 81.)

[WMP07] Nicholas J. Wang, Aqeel Mahesri, and Sanjay J. Patel. Examining ACE analysis reliability estimates using fault-injection. In *Proceedings of the 34th International Symposium on Computer Architecture (ISCA '07)*, pages 460–469. ACM Press, June 2007. doi: 10.1145/1250662.1250719. (Cited on page 76.)

[WP05] Nicholas J. Wang and Sanjay J. Patel. ReStore: Symptom based soft error detection in microprocessors. In *Proceedings of the 35th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '05)*, pages 30–39. IEEE Computer Society Press, June/July 2005. doi: 10.1109/DSN.2005.82. (Cited on page 37.)

[WPS⁺15] Stefan Winter, Thorsten Piper, Oliver Schwahn, Roberto Natella, Neeraj Suri, and Domenico Cotroneo. GRINDER: On reusability of fault injection tools. In *Proceedings of the 10th IEEE/ACM International Workshop on Automation of Software Test (AST 2015)*. IEEE Computer Society Press, May 2015. (Cited on pages 58, 65, and 81.)

[WTLP14] Jiesheng Wei, Anna Thomas, Guanpeng Li, and Karthik Pattabiraman. Quantifying the accuracy of high-level fault injection techniques for hardware faults. In *Proceedings of the 44th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '14)*, pages 375–382, Washington, DC, USA, June 2014. IEEE Computer Society Press. doi: 10.1109/DSN.2014.2. (Cited on pages 64 and 202.)

[WTM13] Vincent M. Weaver, Dan Terpstra, and Shirley Moore. Non-determinism and overcount on modern hardware performance counter implementations. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '13)*, pages 215–224. IEEE Computer Society Press, April 2013. doi: 10.1109/ISPASS.2013.6557172. (Cited on page 183.)

[WTSS13] Stefan Winter, Michael Tretter, Benjamin Sattler, and Neeraj Suri. simFI: From single to simultaneous software fault injections. In *Proceedings of the 43rd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '13)*, June 2013. doi: 10.1109/DSN.2013.6575310. (Cited on pages 65 and 67.)

[Wul13] Daniel Wulfert. Aspektorientierte Implementierung und Bewertung von Fehlertoleranzmechanismen im eingebetteten Betriebssystem eCos. Diploma thesis, Technische Universität Dortmund, March 2013. (Cited on page 208.)

[WZF13] John Paul Walters, Kenneth M. Zick, and Matthew French. A practical characterization of a NASA SpaceCube application through fault emulation and laser testing. In *Proceedings of the 43rd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '13)*. IEEE Computer Society Press, June 2013. doi: 10.1109/DSN.2013.6575354. (Cited on page 61.)

[XL12] Xin Xu and Man-Lap Li. Understanding soft error propagation using efficient vulnerability-driven fault injection. In *Proceedings of the 42nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '12)*, pages 1–12, June 2012. doi: 10.1109/DSN.2012.6263923. (Cited on page 67.)

[XST08] Jianjun Xu, Rui Shen, and Qingping Tan. PRASE: An approach for program reliability analysis with soft errors. In *Proceedings of the 14th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC '08)*, pages 240–247. IEEE Computer Society Press, December 2008. doi: 10.1109/PRDC.2008.30. (Cited on page 41.)

[XTS08] Jianjun Xu, Qingping Tan, and Rui Shen. A novel optimum data duplication approach for soft error detection. In *Proceedings of the 15th Asia-Pacific Software Engineering Conference (APSEC '08)*, pages 161–168. IEEE Computer Society Press, December 2008. doi: 10.1109/APSEC.2008.46. (Cited on pages 41 and 42.)

[XX12] Jun Xu and Ping Xu. The research of memory fault simulation and fault injection method for BIT software test. In *Proceedings of the 2nd International Conference on Instrumentation, Measurement, Computer, Communication and Control (IMCCC '12)*, pages 718–722. IEEE Computer Society Press, December 2012. doi: 10.1109/IMCCC.2012.174. (Cited on page 67.)

[YC80] Stephen S. Yau and Fu-Chung Chen. An approach to concurrent control flow checking. *IEEE Transactions on Software Engineering*, SE-6(2):126–137, March 1980. doi: 10.1109/TSE.1980.234478. (Cited on page 43.)

[YdAL+03] Pedro Yuste, David de Andrés, Lenin Lemus, Juan José Serrano, and Pedro Gil. INERTE: Integrated nexus-based real-time fault injection tool for embedded systems. In *Proceedings of the 33rd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '03)*. IEEE Computer Society Press, June 2003. (Cited on pages 7 and 63.)

[YE09] Doe Hyun Yoon and Mattan Erez. Memory mapped ECC: Low-cost error protection for last level caches. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA '09)*, pages 116–127, New York, NY, USA, 2009. ACM Press. doi: 10.1145/1555754.1555771. (Cited on page 30.)

[YE10] Doe Hyun Yoon and Mattan Erez. Virtualized and flexible ECC for main memory. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 397–408, New York, NY, USA, 2010. ACM Press. doi: 10.1145/1736020.1736064. (Cited on page 37.)

[Yeh96] Ying Chin Yeh. Triple-triple redundant 777 primary flight computer. In *Proceedings of the IEEE Aerospace Applications Conference*, volume 1, pages 293–307, February 1996. doi: 10.1109/AERO.1996.495891. (Cited on pages 37 and 38.)

[Yos13] Junko Yoshida. Toyota case: Single bit flip that killed. http://www.eetimes.com/document.asp?doc_id=1319903, 2013. Retrieved 2015-12-06. (Cited on page 2.)

[YRLG03] Pedro Yuste, Juan Carlos Ruiz, Lenin Lemus, and Pedro Gil. Non-intrusive software-implemented fault injection in embedded systems. In Rogério Lemos, Taisy Silva Weber, and João Batista Camargo, editors, *Proceedings of the 1st Latin-American Symposium on Dependable Computing (LADC '03)*, pages 23–38. Springer-Verlag, October 2003. doi: 10.1007/978-3-540-45214-0_5. (Cited on page 63.)

[YS96] Charles R. Yount and Daniel P. Siewiorek. A methodology for the rapid injection of transient hardware errors. *IEEE Transactions on Computers*, 45(8):881–891, August 1996. doi: 10.1109/12.536231. (Cited on page 58.)

[ZAV04] Haissam Ziade, Rafic A. Ayoubi, and Raoul Velazco. A survey on fault injection techniques. *The International Arab Journal of Information Technology*, 1(2):171–186, 2004. (Cited on pages 4 and 60.)

[ZCM$^+$96] James F. Ziegler, Huntington W. Curtis, Hans P. Muhlfeld, Charles J. Montrose, B. Chin, Michael Nicewicz, C. A. Russell, Wen Y. Wang, Leo B. Freeman, P. Hosier, L. E. LaFave, James L. Walsh, José M. Orro, G. J. Unger, John M. Ross, Timothy J. O'Gorman, B. Messina, Timothy D. Sullivan, A. J. Sykes, H. Yourke, Thomas A. Enger, Vikram R. Tolat, T. S. Scott, Allen H. Taber, R. J. Sussman, W. A. Klein, and C. W. Wahaus. IBM experiments in soft fails in computer electronics (1978–1994). *IBM Journal of Research and Development*, 40(1):3–18, January 1996. doi: 10.1147/rd.401.0003. (Cited on page 27.)

[Zie96] James F. Ziegler. Terrestrial cosmic rays. *IBM Journal of Research and Development*, 40(1):19–39, January 1996. doi: 10.1147/rd.401.0019. (Cited on page 29.)

[ZL79] James F. Ziegler and W. A. Lanford. Effect of cosmic rays on computer memories. *Science*, 206(4420):776–788, November 1979. doi: 10.1126/science.206.4420.776. (Cited on pages 26 and 27.)

[ZLT02] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. SPEA2: Improving the strength pareto evolutionary algorithm for multiobjective optimization. In K. Giannakoglou, D. Tsahalis, J. Periaux, K. Papailiou, and T. Fogarty, editors, *Proceedings of the International Conference on Evolutionary Methods for Design, Optimisation and Control with Application to Industrial Problems (EUROGEN 2001)*, pages 95–100, Barcelona, Spain, 2002. International Center for Numerical Methods in Engineering (CIMNE). (Cited on page 159.)